MASTER EN SCIENCES INFORMATIQUES

Remote Procedure Cali Théorie et implémentation

Nilles, Romain

Award date:
1991

Awarding institution:
Universite de Namur

Remote Procedure Call
Théorie
et
implémentation

Promoteur
Monsieur **Jean Ramaekers**

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maître en Informatique
par

Romain Nilles

Année académique 1990-1991

Facultés Universitaires Notre-Dame de la Paix
NAMUR

Institut d'Informatique

Remote Procedure Call
Théorie
et
implémentation

Romain Nilles

Mémoire présenté en vue de l'obtention du titre de
Licencié et Maître en Informatique
1990-1991

Résumé

Dans ce mémoire nous présentons les aspects théoriques et d'implémentation d'une méthode de répartition d'applications dans un système distribué : le *Remote Procedure Call*, extension du modèle procédural classique. Le premier chapitre tente d'introduire les types, caractéristiques et architectures ainsi que les avantages et désavantages de ce qu'on entend de nos jours par "système distribué". Il traite aussi des réseaux en général, des protocoles de communication adaptés aux systèmes distribués et de quelques aspects de sécurité et de protection y liés. Le deuxième chapitre illustre la facette théorique du RPC, sa philosophie, ses objectifs ainsi que les contraintes d'implémentation et de performance. Enfin le chapitre III propose une implémentation particulière du RPC. Il s'agit du *Network Computing System* (NCS) d'Apollo/HP. L'architecture, les concepts de base et le modèle réseau sous-jacent au NCS y sont analysés et évalués.

Abstract

In this paper we present the different aspects of theory and implementation of a means of partitioning applications in a distributed system : the *Remote Procedure Call* (RPC), an extension to the classic procedure call mechanism. The first chapter introduces the types, characteristics and architectures as well as the advantages and disadvantages of what is considered as a "distributed system". It deals with networks in general, their communication protocols as applied to distributed systems and the related realms of security and protection. The second chapter illustrates the theory of RPC, its philosophy, objectives and constraints of implementation and performance. Chapter III proposes a specific RPC implementation, the *Network Computing System* (NCS). The architecture of NCS, its basic concepts and the underlying network model are analyzed and evaluated.

Que soient ici remerciés tous ceux qui m'ont, d'une manière ou d'une autre, aidé lors de la réalisation de ce travail.

Je pense tout particulièrement à Monsieur Jean Ramaekers, promoteur de mon mémoire, qui m'a guidé et conseillé au cours de cette année académique. Toute ma gratitude à son assistante, Madame Cotet, pour sa disponibilité à mon égard.

Pour ses conseils typographiques et pour l'impression du mémoire je remercie Mr. Fred Hammond, technical marketing documentalist de Goodyear S.A. Luxembourg.

Je tiens enfin à exprimer ma profonde reconnaissance à Monsieur Philippe De Rivet, Monsieur Bernard Martin ainsi qu'aux autres membres du département "Services et Systèmes Distribués" de BULL S.A., Massy, pour m'avoir permis d'effectuer mon stage de fin d'études dans les meilleures conditions. Merci aussi à Eric Lallet, pour m'avoir fait part de ces connaissances détaillées du système UNIX, pour sa patience et son esprit didactique.

Namur, le 31 août 1991

Romain Nilles

Introduction :

Les ***systèmes distribués*** ont atteint depuis quelques années une seuil d'acceptation considérable auprès des milieux universitaires et industriels. Partout, lors de l'installation de nouveaux systèmes informatiques, l'on considère les systèmes distribués pour les nombreux avantages qu'ils sont susceptibles de présenter.

La distribution des données et des traitements est censée amener une meilleure efficacité, sécurité et disponibilité des ressources d'un réseau d'ordinateurs. Ces réseaux sont souvent de nature hétérogène. Les tâches primordiales des concepteurs seront alors de cacher la séparation des ressources et l'hétérogénéité des machines. Il s'agit aussi de garantir la sécurité physique et logique du système et la protection de la confidentialité des données contre les attaques extérieures.

Le chapitre I du mémoire tente de donner une *introduction aux systèmes distribués*, de définir leurs caractéristiques et de discuter de leurs avantages et désavantages. Il y sera question des réseaux et protocoles sous-jacents au système et transparents aux programmeurs et utilisateurs. Le chapitre conclut sur les différents aspects de sécurisation des systèmes distribués et propose quelques esquisses de solution.

Les architectures de communication classiques manifestent une masse sans cesse croissante de protocoles de plus en plus complexes. Dans cette situation le développement et la validation d'applications distribuées risque de devenir une tâche fastidieuse. L'extension du modèle procédural local aux systèmes distribués propose une solution élégante et efficace aux problèmes cités. L'approche ***Remote Procedure Call***, sa philosophie, ses concepts ainsi que les problèmes d'hétérogénéité, de transparence et de performance font l'objet du deuxième chapitre intitulé *Remote Procedure Call - théories*.

Une implémentation RPC particulière et son évaluation seront illustrées au chapitre III : le *Network Computing System* (NCS). NCS trouve sa place dans le système d'exploitation OSF/1, standardisée par l'Open Software Foundation. Il s'agit d'une implémentation portable, puissante et efficace d'un modèle de système distribué proposée par Apollo et Hewlett Packard.

Chapitre I : Introduction aux systèmes distribués 1

Chapitre III : NCS - Network Computing System 59

Annexes

A. Description des interfaces NCS

A.1 Conversation Manager
A.2 Data Replication Manager
A.3 RPC Error
A.4 Location Broker
A.5 Process Fault Manager
A.6 RPC RunTime
A.7 Remote RPC RunTime
A.8 RPC Socket
A.9 RPC UUID

B. Les messages d'erreur de NCS

C. La syntaxe NIDL, fichier YACC

D. NDR, représentation interne des données

E. L'exemple BINOP, binary operations

F. Etudes et mesures

F.1 Fiabilité
F.2 Performance
F.3 Taille des exécutables

G. Le protocole NCA/RPC

G.1 Les 4 FSM client
G.2 Le FSM serveur

Chapitre I - Introduction aux systèmes distribués

1. Les systèmes distribués

L'évolution historique de l'informatique indique ces dernières années une nette tendance vers le remplacement des grandes installations centralisées par des *systèmes distribués*. Partager les ressources informatiques entre plusieurs utilisateurs sur des sites géographiques différents et sur des machines d'architectures divergentes est l'atout majeur de ces nouveaux systèmes.

Dans ce premier chapitre nous allons décrire, sans trop détailler, les diverses catégories et architectures des systèmes distribués, leurs caractéristiques principales et des leurs objectifs de design. Seront exposés ensuite les réseaux informatiques, bases de tout système distribué, et les aspects de sécurité qui y sont liés [CoDo].

1.1 Les types de systèmes distribués

Le but des systèmes distribués est de permettre le partage des ressources en offrant des facilités d'utilisation aussi flexibles et puissantes que dans les *systèmes centralisés*. L'utilisateur devrait avoir l'impression de se trouver en face d'un système centralisé en ignorant le fait que les services lui sont fournis par de multiples machines situées en divers endroits.

Nous distinguons deux types de systèmes distribués.

Les systèmes "*loosely coupled*" [LCS] supposent l'existence de plusieurs machines qui fournissent des services accessibles par une couche software figurant sur chaque machine. Cette couche fera largement usage du réseau pour coordiner les travaux et transférer des données entre les ordinateurs.

Les systèmes "*tightly coupled*" [TCS] ont pour objectif primaire d'atteindre des performances similaires aux systèmes centralisés. A cette fin ils évitent les techniques des réseaux locaux classiques. Ils regroupent plusieurs CPUs qui se partagent une mémoire centrale ou un même espace d'adressage.

1.1.1 Les systèmes distribués "loosely coupled"

Le modèle le plus répandu de systèmes distribués "loosely coupled" est celui où plusieurs ordinateurs mono-utilisateur ou stations de travail accèdent à des données et/ou ressources partagées, mises à disposition par des machines serveurs. Ils utilisent des réseaux locaux à haut débit. Le schéma 1.1 donne une illustration de ce modèle "workstation / server".

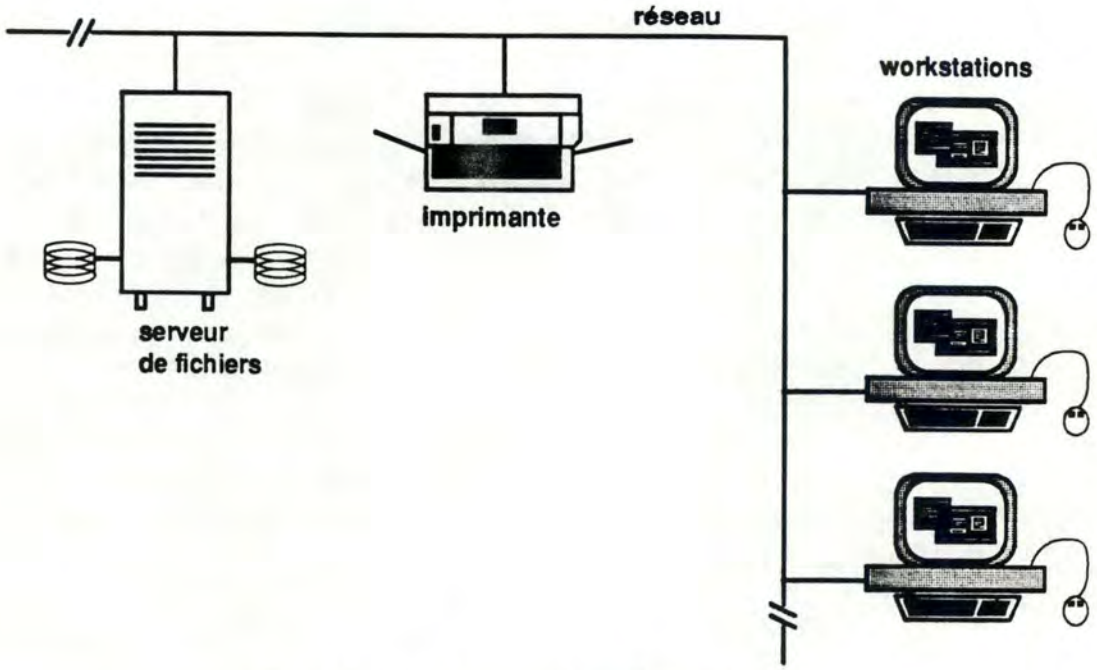

Schéma 1.1 : Petit système distribué

Les *stations de travail* sont des ordinateurs dédiés à un seul utilisateur. Elles sont dotées d'une puissance considérable et souvent aussi d'un écran graphique à haute résolution, ce qui explique leur utilisation pour les applications graphiques hautement interactives (p. ex. applications XWindows).

Afin de garder les avantages des systèmes multi-utilisateurs classiques les stations de travail se trouvent intégrées dans un réseau, avec un accès commun aux ressources.

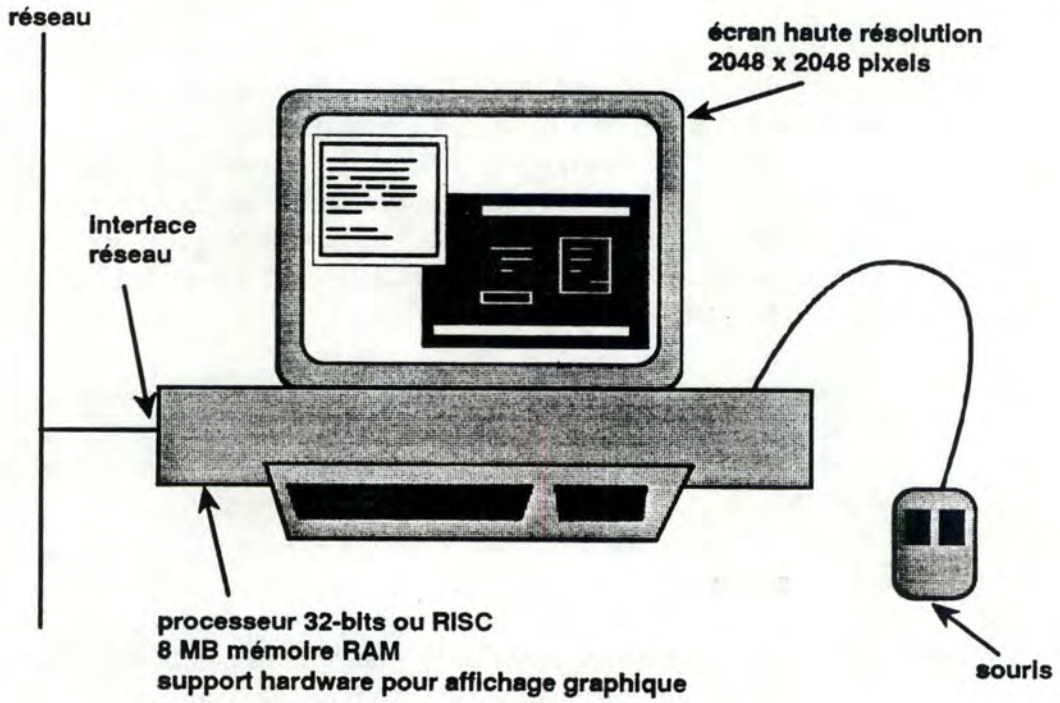

Schéma 1.2 : Station de travail type

Le schéma 1.2 nous montre la configuration et les caractéristiques typiques d'une station de travail.

Le système d'exploitation de référence de la plupart des stations de travail est UNIX [Unix]. Ses principes de mémoire virtuelle paginée entraîne souvent des entrées/sorties disque (*swapping*) . Dans un système avec des stations de travail sans disque la plus grande partie de la charge réseau sera constituée des *swappings*. L'alternative serait d'utiliser un système d'exploitation adapté aux circonstances ou d'équiper chaque machine d'un disque local de taille suffisante. Notons que dans la suite du texte toute expression "système distribué" fait référence aux systèmes distribués "loosely coupled".

Partager des ressources entre utilisateurs implique le hardware ainsi que le software. Certes le partage de périphériques fait réduire les coûts de l'installation entière, mais il ne faut surtout pas sous-estimer les avantages du partage des données dont bénéficient les systèmes multi-utilisateurs centraux.

Une équipe de développement d'applications nécessite le partage des outils de développement comme le compilateur, l'éditeur des liens ou le débogueur ainsi que celui des librairies de procédures. Une fois les modules achevés ils peuvent devenir immédiatement disponibles à tous.

La majorité des applications commerciales nécessitent le partage de données. Prenons l'exemple des guichets de réservation de places d'avion. Ce genre d'application nécessite le partage d'une large base de données.

1.1.2 Les systèmes distribués "tightly coupled"

Les systèmes distribués "tightly coupled" se présentent comme le regroupement d'un certain nombre de processeurs sous le contrôle d'un système d'exploitation unique. Les processeurs ou bien se partagent une mémoire commune physique ou bien une espace d'adressage virtuel unifié avec, entre eux, des connexions à très haut débit. Cette mémoire partagée permettra aux tâches du système de communiquer par le biais de variables et de tables partagées comme cela se fait couramment dans les systèmes centralisés actuels. Les architectures "tightly coupled" comprennent un nombre de plus en plus élevé de processeurs. Etant donné le problème de la bande d'accès à une mémoire partagée l'on dote chaque processeur de son propre cache.

Les "*array processors*" [AP] sont un exemple de système fortement couplé. Ils se présentent comme une intégration de plusieurs unités logiques et arithmétiques (ALU) qui permettent d'effectuer des calculs matriciels et d'autres opérations régulières sur un ensemble de données. Ces machines sont appelées ordinateurs SIMD (*Single Instruction Multiple Data*) puisque une instruction machine provoque le traitement parallèle sur toute une série de données.

D'autres architectures [OPA] comme celle du "*dataflow*" ont vu le jour. L'idée est de déceler le parallélisme irrégulier au niveau des instructions de la machine. Le calcul d'expressions mathématiques, suite à leur décomposition en instructions parallèles, est un des domaines d'application de la théorie du "dataflow".

3

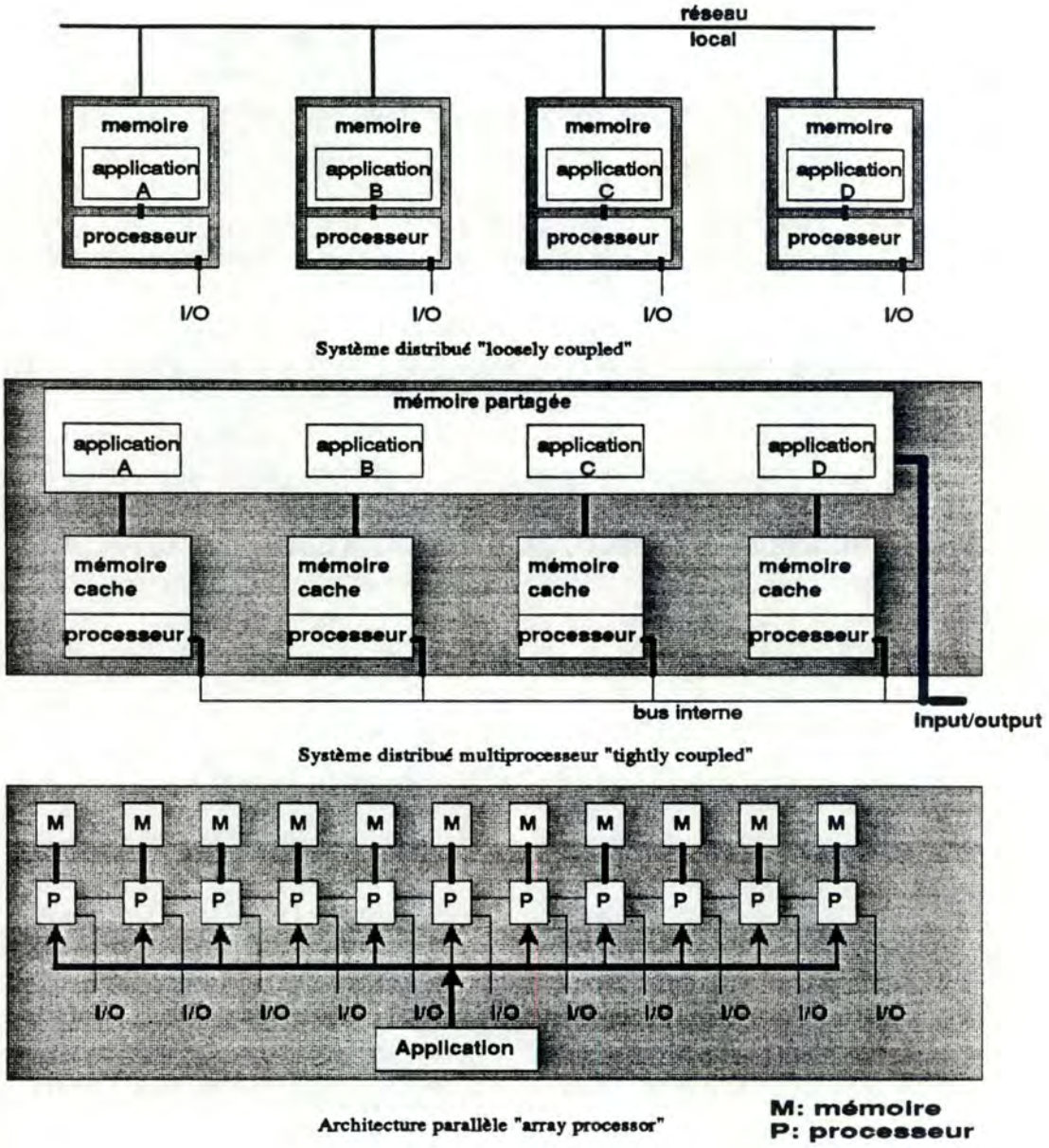

Schéma 1.3 : Types de systèmes distribués

1.2 Les motivations qui ont menés vers les systèmes distribués

Pourquoi cette percée des systèmes distribués ? Quels sont les besoins techniques et les pressions économiques sur lesquels se base leur succès ?

. La **chute du prix des processeurs** VLSI et des mémoires vives

. La **disponibilité de technologies de réseau** performantes et cela à coût modéré

. Le **service interactif** fourni par les systèmes centralisés n'est guère satisfaisant, avec des temps de réponse longs, des interfaces utilisateur archaïques et des difficultés constantes de reconfigurer le hardware et le software aux besoins changeants des divers groupes d'utilisateurs

4

. L'existence d'une série de **nouvelles applications** auxquels les systèmes classiques ne peuvent plus faire face : environnements graphiques évolués, algorithmes de recherche complexes, traitement de texte à fontes multiples,CAD...

La Loi de Grosch [Grosch], qui affirme que la puissance de calcul d'un ordinateur est plus ou moins proportionnelle au carré de son coût, est-elle encore valable ? Selon cette loi un ordinateur central d'un million de francs serait-il encore plus puissant que deux machines de 500.000 francs. L'arrivée de stations de travail puissantes de SUN, HP et DEC ainsi que des "micro-ordinateurs" Intel '486 semblent mettre bien en doute cette logique.

La capacité de calcul n'est pas le seul facteur à prendre en compte. Les nouvelles applications posent une charge énorme sur les systèmes multi-utilisateurs de façon à ce que tôt ou tard la surcharge sera inévitable. La mise en place de réseaux de machines performantes mono-utilisateurs, avec des temps de réponse courts et des dispositifs de mise à jour instantanée des écrans graphiques, est un prérequis pour les applications informatiques actuelles.

1.3 Les caractéristiques des systèmes distribués

Les frontières entre les systèmes centralisés et distribués ne sont pas définis de façon précise. Il n'existe pas une caractéristique ou règle unique qui permet de définir les systèmes distribués.

[LeLann] note comme objectifs majeurs d'un système distribué son **extensibilité**, sa **disponibilité** accrue et le **partage optimal des ressources**. Ces objectifs se basent sur l'existence d'une multiplicité de composantes de même type, l'interconnexion de processeurs, la "*transparence*", l'absence de structures de contrôle hiérarchiques et l'existence de processus tournant dans des espaces d'adressage disjoints et communiquant par messages.

Que faut-il entendre, dans ce contexte, par "*absence de structures de contrôle hiérarchiques*" ? Les systèmes de réservation de tickets ainsi que le système BANCONTACT, où un ordinateur central contrôle toutes les machines du réseau, sont-ils vraiment à considérer comme systèmes distribués ? On leur réserve plutôt le nom de systèmes "basés sur réseau" (*network based systems*).

Les auteurs du ANSA Reference Manual [ANSA] définissent les conséquences de la distribution en termes de ***séparation*** et de ***transparence*** .

1.3.1 La séparation

La séparation est une propriété inhérente des systèmes distribués. Elle implique l'utilisation de moyens de communication et de techniques de management de réseaux.

5

Elle permet :

- l'exécution parallèle de programmes
- l'encapsulation des fautes de certaines composantes
- le recouvrement de pannes partielles avec possibilité de faire tourner le système à moindre performance
- l'utilisation de procédés d'isolation par verrous comme méthodes de renforcement de la sécurité
- l'accroissement et le rétrécissement du système par simples ajouts ou retraits de composantes

1.3.2 La transparence

La *transparence* a comme objectif de ***cacher la séparation*** aux yeux des utilisateurs et des programmeurs de façon à ce que le système soit perçu comme un tout et non comme un assemblage d' ordinateurs et de moyens de communication.

[ANSA] définit huit types de transparence :

- la ***transparence d'accès***, qui permet l'accès aux objets lointains (fichiers, périphériques, ordinateurs ...) par des opérations ayant la même syntaxe et la même sémantique que leurs contreparties locales

- la ***transparence de localisation***, qui permet l'accès aux objets lointains sans connaître leur emplacement exacte dans le réseau

- la ***transparence de parallélisme***, qui permet à plusieurs utilisateurs de travailler en parallèle sur des données partagées sans interférence quelconque entre eux

- la ***transparence de réplication***, rendant possible l'existence cachée de multiples instances du même objet dans la perspective d'augmenter la fiabilité et la disponibilité du système

- la ***transparence aux pannes*** , qui cache les pannes de certaines composantes du système et permet aux applications de terminer leur travail en dépit d'erreurs du hardware ou du software

- la ***transparence de migration***, qui permet le mouvement d'objets lointains à l'intérieur du système sans affecter les processus en cours

- la ***transparence de performance***, qui rend possible la reconfiguration dynamique du système suite à une modification de sa charge

- la ***transparence d'échelle***, qui facilite l'extension du système ou des applications sans changement aucun de la structure du système ou des algorithmes des applications existantes

6

1.4 Discussion

Le remplacement des systèmes existants par des systèmes distribués est accompagné d'une série d'avantages et de désavantages [CoDo].

1.4.1 Les avantages des systèmes distribués

. **Temps de réponse prévisible.** Les systèmes distribués conviennent aux utilisateurs faisant tourner plusieurs applications fortement interactives qui nécessitent des puissances de calcul raisonnables sans toutefois être du type "*number crunching*". Des stations de travail avec de puissances allant jusqu'aux 10 MIPS se prêtent particulièrement bien à cette tâche.

. **Extensibilité.** Le manager d'un système distribué peut facilement en étendre les capacités en fonction de la demande des utilisateurs et ceci sans devoir remplacer les composantes déjà existantes. Les systèmes distribués peuvent aller de deux machines plus serveur de fichiers à une centaine de stations de travail avec plusieurs serveurs de fichiers, d'imprimantes etc. A la demande des utilisateurs et selon le budget de l'entreprise concernée l'on pourra alors ajouter successivement des serveurs ou des stations, la seule limite étant la largeur de la bande du support de communication.

. **Partage de ressources.** L'utilisation du réseau de communication permet de partager les ressources du système entre toutes les machines reliées.

. **Réplication.** En gardant plusieurs copies des objets du système (logiciels, données ou périphériques) sur des machines différentes on garantit un haute fiabilité et un accès rapide aux ressources disponibles.

. **Disponibilité continue.** L'arrêt d'un composant du système n'entraîne pas nécessairement l'arrêt du système entier. Seul les processus qui en ont fait usage sont concernés. Ils peuvent migrer sur une autre machine ou bien essayer de trouver les données ou périphériques nécessaires à leur fonctionnement en un autre endroit.

1.4.2 Désavantages des systèmes distribués

. **Allocation moins flexible des ressources mémoire et CPU.** Dans un système centralisé toutes les ressources mémoire et CPU peuvent être alloués à un processus sous le contrôle d'un seul système d'exploitation. Dans le cas de systèmes distribués c'est la taille de la mémoire et la puissance du CPU des stations de travail qui déterminent la taille maximale de la tâche qui peut être exécutée.

. **Dépendance de la fiabilité et de la performance du réseau.** Une panne de réseau entraîne inévitablement l'arrêt d'une partie de l'activité du système distribué. Une surcharge du réseau fait dégrader les temps de réponse aux utilisateurs. Notons la survenance assez rare de pannes de réseau et les recherches incessantes dans la construction de réseaux plus fiables.

. **Les lacunes dans la sécurité.** Afin de faciliter l'extensibilité du système les interfaces software du système sont librement disponibles aux clients. Chacun qui possède un accès au support de communication a ainsi accès aux interfaces de tous les serveurs.

7

Cette architecture "*ouverte*" est très accueillante aux développeurs et aux ingénieurs systèmes, mais l'installation de systèmes de protection s'est avéré indispensable pour éviter l'accès non-autorisé, accidentel ou intentionnel, aux ressources du système ainsi qu'aux fichiers confidentiels.

1.5 Les différentes architectures distribuées

Décrire l'architecture d'un système distribué revient à illustrer ses différentes composantes software et hardware, - le type de machines, leur localisation dans le réseau et la localisation de l'exécution des diverses applications système et utilisateurs -, ainsi que les relations qu'il y a entre celles-ci.

Dans ce cadre on verra trois modèles différents :

. **le modèle "station de travail / serveur" (*workstation/server*)**

. **le modèle "pool de processeurs" (*processor pool*)**

. **le modèle "intégré" (*integrated*)**

Les modèles illustrés ci-dessous sont tous basés sur l'existence d'un software système distribué modulaire. En plus des tâches spécifiques d'un système d'exploitation standard ce software doit fournir des services supplémentaires de communication. Ces services, au lieu d'être regroupés dans un noyeau sur une machine centrale, peuvent se trouver éparpillés à travers tout le réseau.

Le noyeau simplifié de ce système d'exploitation se trouvera sur chaque machine du réseau. Il ne comprend plus de gestion de conflits entre utilisateurs sur ces ordinateurs mono-utilisateurs et décharge de la gestion de fichiers et des autres périphériques sur des serveurs spécifiques prévus à cet effet. De nombreuses recherches ont été faites pour aboutir à des noyaux plus légers (*lightweight kernels*) basés sur des mécanismes puissants de communication inter-processus ainsi que sur les processus légers (*threads*). Un objectif important est d'éviter le *overhead* résultant de chaque changement de contexte (task switching) qu'on trouve dans les noyaux traditionnels comme en UNIX [Unix]. Au lieu d'avoir des processus isolés qui communiquement difficilement, entraînent des *swappings* coûteux et réduisent ainsi le throughput du système à chaque *task switch*, un noyeau léger disposant de petits processus qui se partagent une mémoire commune et qui communiquement via des tables et des variables partagées semble plus approprié aux systèmes distribués.

1.5.1 Le modèle "workstation / server"

Cette architecture [SUN,XEROX,NCA] nous propose plusieurs ordinateurs mono-utilisateurs, stations de travail, ayant tous accès au réseau. Des serveurs fournissent des services d'accès aux périphériques communs (serveurs d'imprimantes) et aux données partagées (serveurs de fichiers) ainsi que les autres fonctions normalement offerts par les systèmes d'exploitation centraux (serveurs d'authentification des utilisateurs, serveurs de noms ...).

8

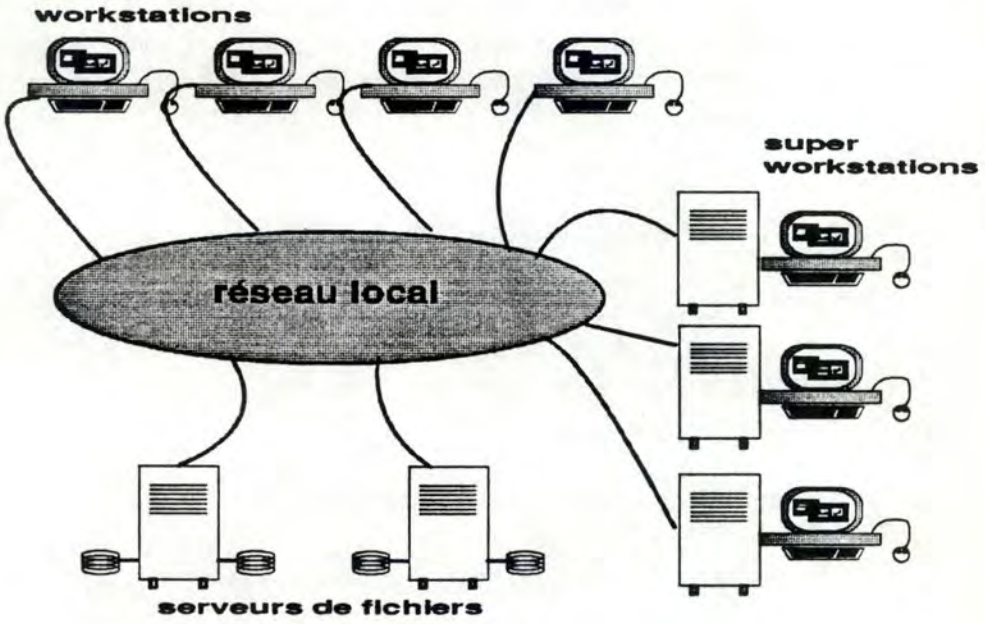

Schéma 1.4 : Modèle "workstation/server"

1.5.2 Le modèle "pool de processeurs"

Le schéma 1.5 met en évidence les éléments suivants : des *terminaux simples*, des *PADs*, les *processeurs pool* ainsi que quelques serveurs de fichiers [LCS 4.]. Les terminaux sont reliés aux PADs, concentrateurs de terminaux qui fournissent la liaison d'un terminal à un processeur pool spécifique. Les serveurs de fichiers gèrent les données partagées par les terminaux vu que ceux-ci ne disposent pas de disque à eux. Les *processeurs pool* sont des mini- ou micro-ordinateurs de puissance variable sans terminaux directement connectés. Lors de la mise en marche d'un terminal l'utilisateur se trouve en face d'un serveur d'allocation de processeur. L'utilisateur spécifie le type de processeur dont il a besoin et ce serveur lui allouera un processeur disponible pour la durée d'une session. Un autre serveur s'occupera ensuite de charger le système d'exploitation sur le processeur alloué et de passer en fin de compte la main à l'utilisateur qui va interagir directement avec le processeur en question.

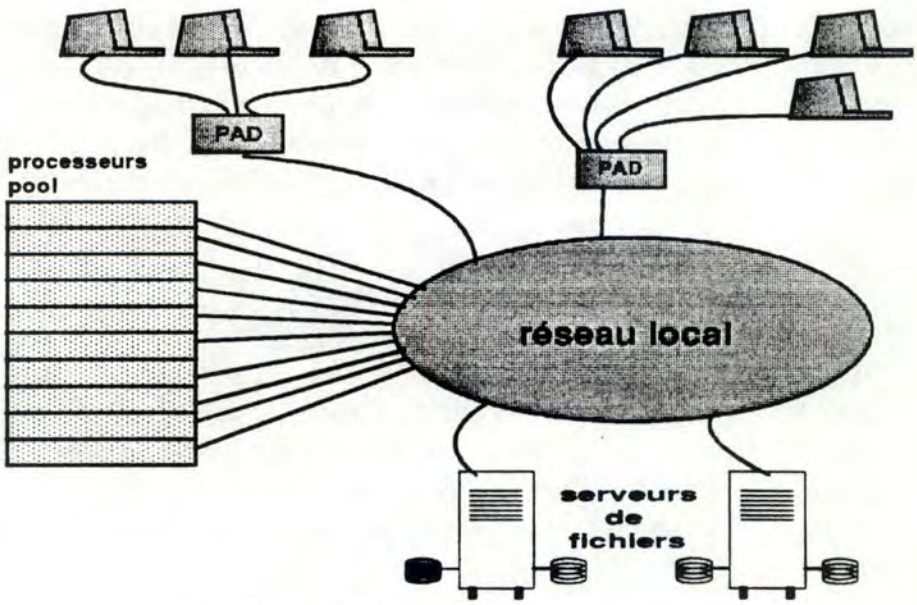

Schéma 1.5 : Modèle "processor pool"

Les avantages de ce modèle par rapport au modèle précédent sont multiples:

. ***Meilleure utilisation des ressources.*** Le nombre de processeurs requis pour supporter une population d'utilisateurs donné est proportionnel au nombre d'utilisateurs dans le système à un moment. Le modèle "workstation/server" nécessite un nombre bien plus élevé d'ordinateurs puisque certaines stations peuvent, temporarirement, être physiquement inaccessibles.

. ***Flexibilité.*** Les services du système peuvent être étendus sans installer de nouveaux ordinateurs. Les *processeurs pool* peuvent jouer le rôle de serveurs qui s'occupent de gérer la charge supplémentaire du système.

. ***Compatibilité.*** Des applications tournant sur systèmes centralisés peuvent être repris moyennant des modifications mineures tandis que les workstations nécessitent des application spécifiques pour exploiter leurs capacités graphiques.

. ***Processeurs hétérogènes.*** Toute un gamme de machines différentes jouent le rôle de processeurs pool, ce qui permet aux utilisateurs d'utiliser des processeurs de performances et de compatibilité différentes.

Néanmoins ce système ne satisfait toujours pas les besoins énormes des applications graphiques "haut de gamme", sachant que des quantités énormes d'informations graphiques auront à passer par le réseau.

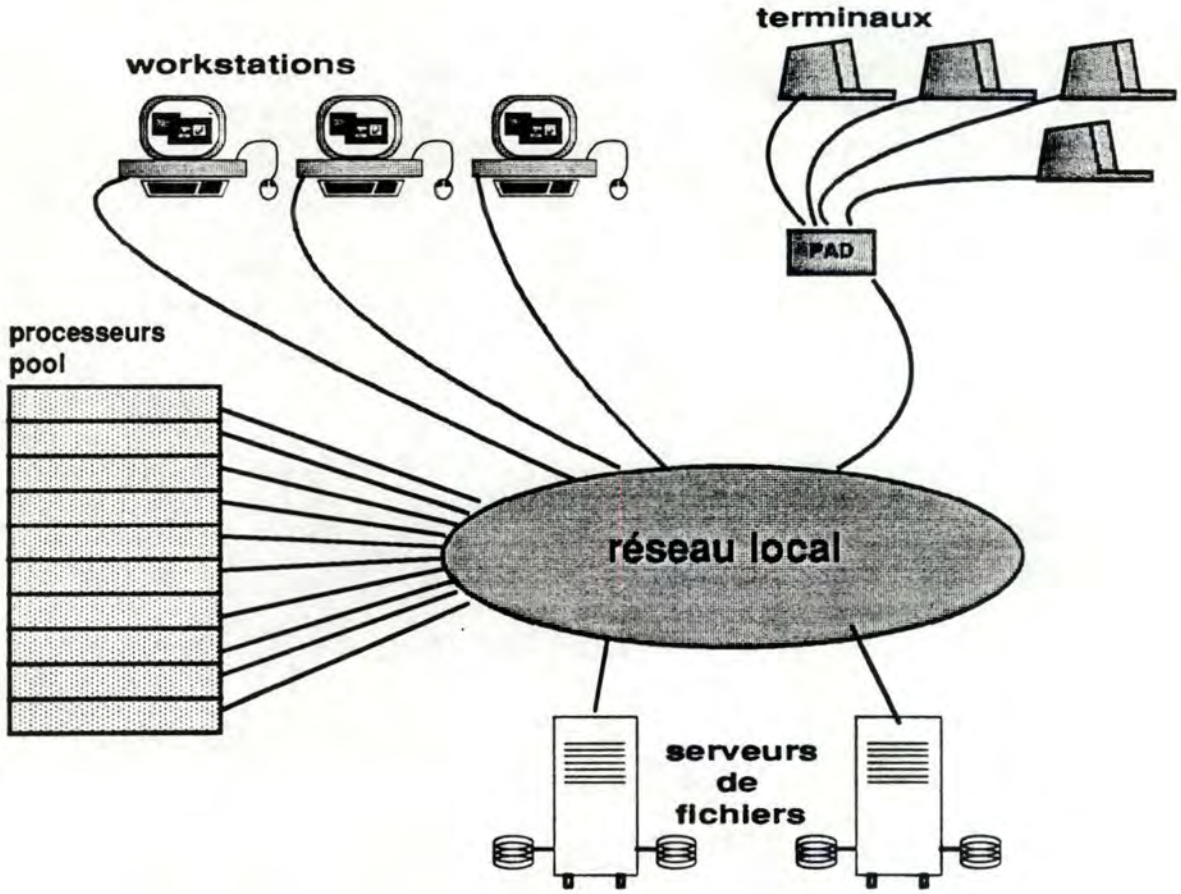

Schéma 1.6 : Modèle hybride

10

Des systèmes *hybrides* [LCS 3.] regroupent les avantages des deux architecture présentées ci-dessus :

. ***Adéquation de la puissance aux besoins des utilisateurs.*** Une application ne trouvant pas les ressources nécessaires sur les stations de travail sera dès lors exécutée sur un processeur pool. Celui-ci pourra éventuellement rediriger ses entrées/sorties vers la station de travail utilisée.

. ***Exécution parallèle.*** L'utilisateur n'est pas limité à utiliser un seul processeur pool. Des applications gourmandes en CPU peuvent être réparties entre plusieurs processeurs pool.

. ***L'accès via des terminaux.*** Les terminaux ne sont pas exclus du système comme dans le modèle workstation/server. En effet certaines applications ne nécessitent point les puissances graphiques des stations de travail et se contentent de terminaux bon marché.

1.5.3 Le modèle intégré

Ce modèle [LCS 1. 2.] est constitué de stations de travail, de serveurs, de terminaux liés aux processeurs pool via PAD ainsi que des terminaux liés directement à des ordinateurs. L'innovation par rapport aux architectures précédentes est la présence d'un operating system intégré qui est distribué sur toutes les machines du réseau. Toutes les machines ont leur autonomie mais se partagent les données en utilisant un nommage global au niveau du réseau. Du moment qu'un processus est destiné à tourner le système sélectera la machine concernée, localisera le programme exécutable et le chargera sur cette machine pour l'exécuter. Locus [LCS 1.], ainsi que UNIX [Unix] accompagné de NFS (*Network Filing System*) [SUN] sont deux exemples de systèmes actuellement en usage.

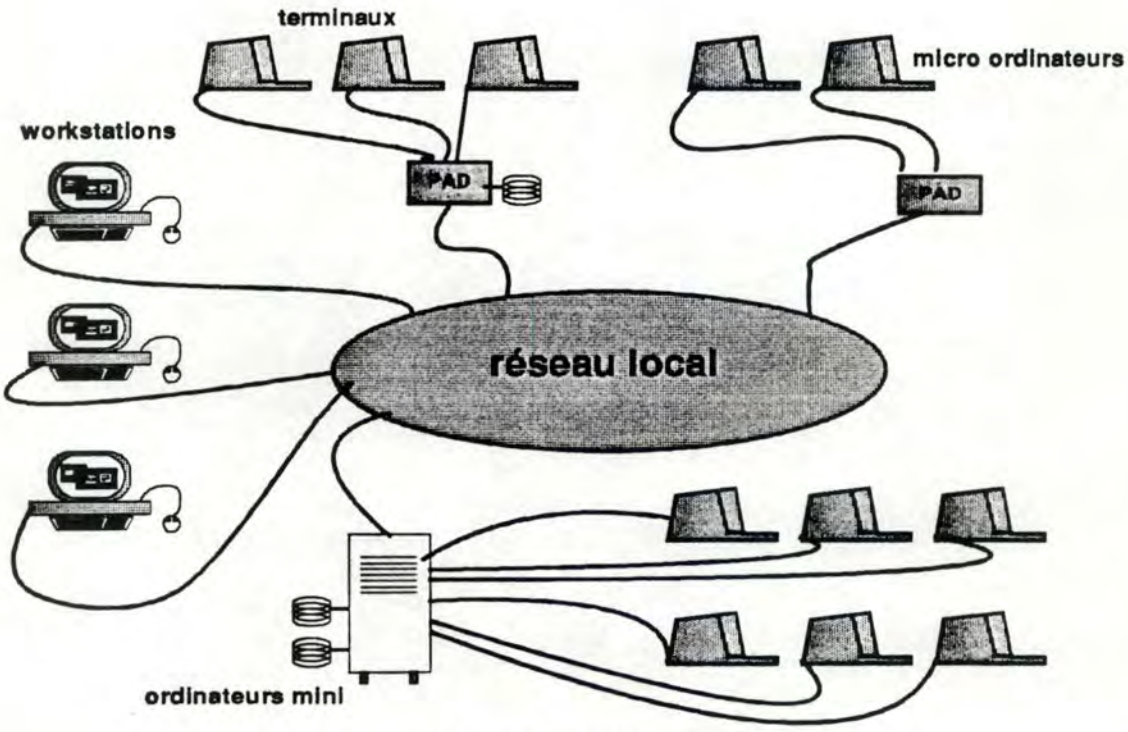

Schéma 1.7: Modèle intégré

1.6 Les objectifs de design des systèmes distribués

L'objectif primordial lors du design des systèmes distribués est de rendre son utilisation et le développement d'applications tout à fait *transparent* aux utilisateurs et aux programmeurs, c'est-à-dire de cacher entièrement la distribution de ses diverses composantes. L'utilisateur devrait avoir l'impression de se trouver en face d'un système entier tout en ignorant l'emplacement des diverses ordinateurs et périphériques.

En plus de cela l'utilisateur est en droit de demander au système d'exécuter ses applications de façon *cohérente* et *efficace*.

1.6.1 La transparence de localisation.

La transparence se situe à plusieurs niveaux.

. **Niveau 0** : A ce niveau l'utilisateur voit le système comme la composition de machines qui peuvent communiquer entre elles à travers des outils particuliers tels que le login remote, l'exécution remote de processus ainsi que le transfert de fichiers. Ces outils sont construits directement sur le software réseau qui permet le passage de message entre les divers processus. Pour supporter ces communications des standards internationaux (modèle OSI) ou *de facto* (TCP/IP du DOD) ont vu le jour.

. **Niveau 1** : Ici certaines applications tentent de cacher l'environnement physique du système distribué. Lors de l'utilisation de ces applications l'utilisateur n'est pas conscient que derrière leurs entrées/sorties se cachent la coopération de plusieurs machines du réseau. Ces applications sont constitués comme des véritables systèmes distribués avec leur propre mécanisme de localisation des données et des serveurs. Prenons comme exemple l'application *rwho* en UNIX [Unix] qui donne en résultat des informations sur tous les utilisateurs du réseau. Il en est de même pour la messagerie électronique. La transparence au niveau 1 requiert des protocoles-application plus évolués qui, en partie ont été l'objet d'efforts de standardisation (X400, FTAM, FTP, SMTP ...) . En l'absence de ces normes le portage des applications existantes vers des environnements différents devient une tâche fastidieuse.

. **Niveau 2** : Le niveau 2 prévoit l'existence de serveurs globaux fournissant des services généraux de partage de données et de ressources hardware. Les programmeurs d'applications s'adresseront à ces serveurs en ignorant la configuration physique du système distribué. Les serveurs de fichiers tels que le Network Filing System (NFS) ainsi que certaines implémentations de protocoles de *Remote Procedure Call*, illustrent bien ce type de transparence. Le système Locus permet de rendre transparent non seulement l'emplacement des fichiers mais aussi l'emplacement de processus en exécution. Le but des serveur de gestion de fenêtres (X Windows ou NeWs) est de créer une transparence entre l'application et l'utilisateur. Ils offrent des services de gestion d'écrans locaux où lointains sans que l'on ait recours à une modification quelconque des applications graphiques.

1.6.2 La cohérence.

Un système tourne de façon cohérente du moment que son comportement est prévisible. Des manques de cohérence peuvent se présenter lors de la mise à jour de fichiers, de bases de données ou de l'écran visuel.

12

- La réaction suite aux pannes

Une panne dans un système centralisé entraîne le plus souvent l'arrêt complet de celui-ci. A l'inverse les systèmes distribués réagissent différemment aux pannes. Une panne d'un composant n'arrête pas tout et des traitement partiellement complétés en sont le résultat. Cette caractéristique se retrouve souvent sous le nom de *"independent failure mode "*.

- L'incohérence des bases de données

Deux problèmes se posent ici :

i) Comment permettre l'usage concurrentiel d'une base de données unique par plusieurs applications ?

Pour résoudre ces conflits et éviter les problèmes du "*lost update*" (mise à jour perdue) le système doit fournir des mécanismes de verrouillage, de fichiers entiers où de leurs enregistrements, permettant la séquentialisation de la lecture et de l'écriture.

ii) Comment réagir à la survenance d'une panne du système (workstation, serveur, réseau, software ...) au moment de la mise à jour de la base de données ?

Un service de *transactions atomiques* sur les fichiers garantira la consistance de la base de données à tout moment. Une *transaction* étant un regroupement d'instructions de lecture/écriture considérés comme une instruction atomique, ce qui évite toute mise à jour partielle.

- La cohérence de l'interface utilisateur

L'interface utilisateur devrait, en principe, à tout instant donner une image de l'état des applications utilisateur . Pour cela l'écran doit être mis à jour suite à chaque événement, chaque interaction avec l'utilisateur. Le délai interactif (*interactive delay*) entre la survenance de l'événement et la mise à jour de l'écran est à réduire à un minimum. La quantité de données à transmettre pour le raffraichissement d'un écran bitmap est gigantesque et nécessite ou bien des lignes de communication à très haut débit ou bien un mapping direct de la mémoire de la workstation et du dispositif de visualisation.

Une façon élégante pour résoudre ce problème de latence est de considérer des serveurs graphiques qui fournissent des interfaces procéduraux pour des fonctions graphiques de haut niveau. X Windows [SchGet] et NeWs [News] sont deux représentants de cette classe de serveurs.

1.6.3 L'efficacité

Plusieurs facteurs, autres que la puissance CPU et la capacité RAM, influent sur l'efficacité d'un système distribué.

- **Le temps de réponse.** Le temps de réponse dans les systèmes centralisés est lent et fluctuant en fonction de la charge. L'allocation d'un ordinateur par utilisateur rend le temps de réponse beaucoup plus court et surtout plus régulier.

- **L'extensibilité.** Un système distribué doit permettre une extension facile et peu coûteuse. Ajouter des composantes au système telles que des nouvelles stations de travail, des serveurs ou moyens de communication (gateways, répéteurs ...) ne doit nullement interrompre les utilisateurs actuels.

- **La fiabilité, la tolérance aux pannes et le recouvrement.** La structuration des systèmes distribués fournit déjà une protection contre le shutdown total dû à l'arrêt d'une composante. Des mécanismes de réplication des ressources et des données stratégiques et d'autres technologies logicielles garantissent la tolérance aux pannes et le recouvrement du système. Des serveurs de fichiers transactionnels remettent le système de fichiers en un état consistent après un shutdown partiel ou complet.

2. Les réseaux et les protocoles

Les systèmes distribués sont bâtis sur l'utilisation d'un moyen de communication qui regroupe plusieurs ordinateurs en un réseau. La suite illustrera les divers types de réseaux, leurs principes de base et leurs protocoles de communication [Tanenbaum].

2.1 Les types de réseaux

Les réseaux peuvent être de deux sortes : locaux ou à longue portée (*local - / wide-area networks*).

Les réseaux à longue portée consistent en une collection de circuits de communication entre plusieurs ordinateurs spéciaux connus sous le nom de *Packet Switching Exchanges* (PSE). Chaque noeud du réseau possède son propre PSE qui s'occupe à gérer le trafic informationnel qui le traverse. Le réseau utilise pour la transmission de messages la méthode *store-and-forward*. Celle-ci consiste à stocker les messages destinés au noeud local et de relayer les autres vers le noeud suivant sur le chemin qui leur reste à parcourir. La communication se fait par des lignes publiques/ louées/privées, des circuits à bande large comme les fibres optiques ou bien par satellite. Notons la mise à disposition au public de services de réseaux digitaux par des organismes publics ou privés comme le réseau DARPA aux USA, PSS en Grande Bretagne ou TRANSPAC en France.

14

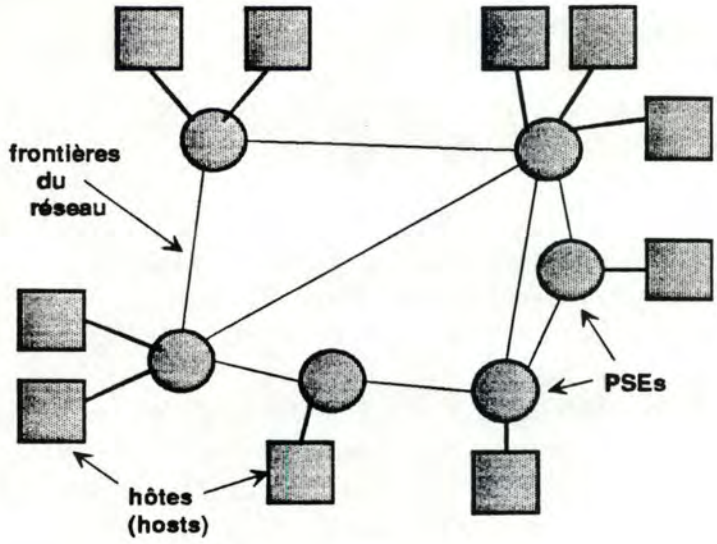

Schéma 2.1 : Réseau "wide area"

Il devrait être clair que les temps de transmission de ce type de réseau ne sont guère suffisants pour permettre une utilisation majeure dans le cadre de systèmes distribués. C'est pourquoi la plupart des systèmes distribués sont basés sur un réseau local.

Les réseaux locaux interconnectent plusieurs machines séparées par des distances plutôt courtes, n'excèdant rarement le kilomètre. Si les distances permises sont excédés des répéteurs sont insérés ont les différents segments du réseau local. Le mode de communication est le *broadcast*, c'est-à-dire qu'un message émis est transmis à tous les noeuds du réseau. Chacun des ordinateurs hôtes possède une interface vers ce réseau et gère son trafic en identifiant et en sauvegardant les informations qui le concernent. Des technologies de communication utilisés sont le *bus EtherNet* de Xerox, le *slotted ring* de l'université de Cambridge et le *token ring* d'IBM [Tanenbaum].

2.2 Les principes d'interconnection de machines

2.2.1 Les paquets

L'unité logique de transmission dans un réseau est le message. Puisque un message peut être d'une taille arbitraire on procède à sa division en plusieurs paquets. Cette découpe se fait dans la *couche transport* du logiciel de communication. Les paquets contiennent, en plus des informations utiles, des informations de gestion comme les adresses de machines origine et destination. Ces paquets sont de taille fixe pour :

1. permettre aux ordinateurs d'allouer une zone tampon suffisante (pour le paquet le plus grand).

2. permettre le partage du réseau et ne pas soumettre les machines à des attentes exagérées suite à la transmission de paquets excessivement longs.

15

2.2.2 Les protocoles

Un *protocole* est défini comme un ensemble de règles et de conventions strictes qui permettent à deux logiciels, sur deux machines différentes, de communiquer. Le protocole décrit le format et la taille des données transmises ainsi que le déroulement des différents étapes lors de la transmission des paquets. Ainsi le rôle d'un protocole de transport est de transmettre des messages d'une certaine longueur d'une origine vers une destination. Pour cela il coupera les messages en d'autres unités qui conviennent à un protocole de niveau inférieur, dans notre cas il s'agit du protocole réseau.

Le logiciel réseau est en principe constitué d'une hiérarchie de couches (*layers*), de modules. Une couche est logiquement vu comme un processus qui dialogue directement avec une autre couche sur une autre machine. En respect des principes de modularisation un protocole ne transmet pas directement les messages à son interlocuteur. Il se chargera de les adapter et ensuite de les passer à un protocole de niveau inférieur et ceci jusqu'au moment où le protocole résultant sera tellement simple qu'il puisse être transmis par une "interface hardware" de réseau. Le procédé inverse se déroulera sur la machine cible où le hardware capte un signal, le passe à une couche supérieure jusqu'au protocole initial.

Une architecture de communication générale pour les réseaux a été présentée et standardisée par l'ISO (*International Standards Organization*) au début de années '80. Il s'agit du **modèle de référence OSI** (*Open Systems Interconnection*) qui reprend une architecture à 7 couches et la définition des protocoles relatifs. Le paragraphe suivant va brièvement illustrer ce modèle. Pour plus d'informations le lecteur est avisé de consulter [Tanenbaum].

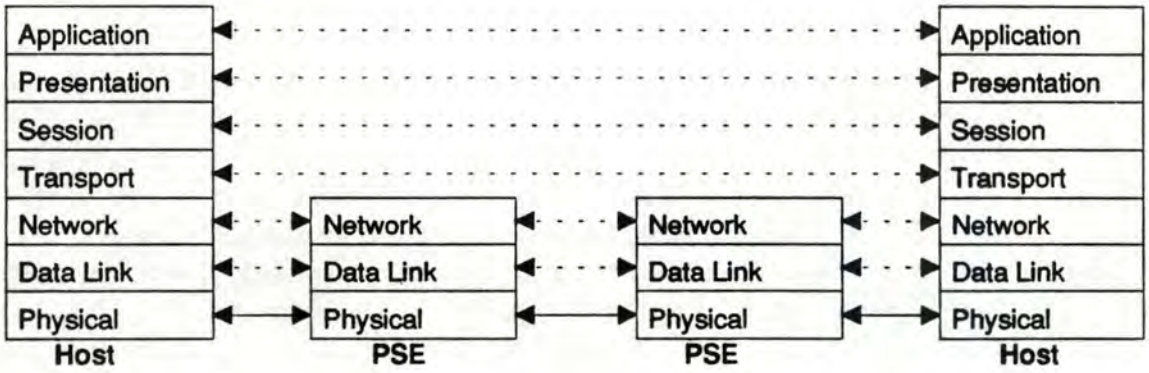

Schéma 2.2 : Modèle de référence ISO/OSI

2.2.3 Le modèle ISO-OSI

Ce paragraphe reprend de haut en bas les 7 couches du modèle OSI et leurs fonctions principales.

- ***Le niveau application***. Sa fonction est de permettre à deux applications de communiquer entre eux. Chaque type d'application nécessitera pour cela son *protocole d'application*. A ce niveau on trouve entre autres le transfert de fichiers ou la messagerie électronique. Etendre le niveau application signifie y introduire l'application elle-même suivie de son propre protocole-application. Il serait toutefois bien plus pratique de pouvoir disposer d'une suite

16

limitée de protocoles sur lesquels pourraient se base toutes les applications. Or cette forme d'extension n'était point prévue par l'ISO. Une solution souvent adoptée est alors de fournir un protocole unique de *Remote Procedure Call* (RPC) utilisé par toutes les applications.

- ***Le niveau présentation***. Sa tâche est de fournir une représentation unique des données transmises pour toute une gamme de machines hétérogènes. Il se chargera de chiffrer, de compresser et de transformer les messages transmis en un format externe (EDR, *External Data Representation*) et les opérations inverses.

- ***Le niveau session***. Il établit des *connexions virtuelles* et s'occupe des synchronisations entre processus qui communiquent en tournant sur des machines différentes.

- ***Le niveau transport***. La couche transport fournit un service de transport, de transmission de messages entre deux *ports* (aussi appelés *sockets*). Un port est défini comme un point de destination de messages. La couche transport délivre alors des messages à destination d'une adresse transport qui est composée d'une adresse réseau, d'un identifiant d'une machine et d'un port à l'intérieur de celle-ci. Deux modes de communication sont disponibles : le *circuit virtuel*, canal logique entre une origine et une cible avec transfert fiable des données, et le *datagramme*, protocole non-connecté mais sans garantie de fiabilité des transmissions.

Vu la fiabilité des réseaux locaux le mode non-connecté semble être le compromis idéal. Les quelques erreurs résiduelles seront alors pris en charge par les niveaux supérieurs ou par le protocole RPC.

- ***Le niveau réseau***. Il s'agit ici de *router* des paquets de données en un format acceptable dans un réseau spécifique. Notons que dans un réseau local ce routage n'est point nécessaire ce qui explique souvent l'absence de cette couche.

- ***Le niveau datalink***. Il est responsable de la transmission sans erreurs entre ordinateurs connectés physiquement. Dans les réseaux à longue portée il s'agit de communications entre les PSE et dans le cas des réseaux locaux des communications entre hôtes locaux.

- ***Le niveau physique***. Cette couche est constituée du hardware qui est le réseau et des circuits physiques entre ordinateurs. Il s'occupe de transmettre des unités d'information "de base" (bits,bytes,mots) au hardware qui lui s'occupera de les adapter (moduler) au moyen de communication. L'établissement de standards hardware et software et la tendance vers la signalisation digitale (ISDN) réduisent la complexité de cette couche physique.

2.3 Les protocoles pour les SD

2.3.1 Les protocoles légers

La transmission de messages se compose d'un *temps de latence*, pour initier la communication et passer par les différentes couches du software, et d'un *temps de transmission* pour transmettre les informations finales par un canal de communication. Le modèle OSI qui crée à chacune de ses couches un *overhead* supplémentaire se heurte aux besoins de performance des systèmes distribués. Comment permettre aux processus clients de communiquer avec les serveurs en un minimum de temps, de l'ordre de micro- ou millisecondes avec des taux de transmission d'au moins 1 mégabits/seconde ?

17

Pour minimiser le temps de latence, temps passé dans le software de communication, il faut minimiser les exécutions excessives des divers modules - le *task switching* et les synchronisations font perdre du temps - ainsi que les données qui transitent entre ceux-ci. Dans un réseau local "fiable" on pourrait regrouper une partie de la fonctionalité des couches supérieures, de transport, de session et de présentation au niveau application. Ce serait le *protocole de Remote Procedure Call* dont on a déjà parlé. Il gérera les confirmations, la conversion entre les représentations des données et la détection/correction d'erreurs lors de la communication.

Puisque les réseaux locaux ne font pas de routage de paquets - la couche de réseau devenant ainsi redondante - le software des systèmes distribués se présente souvent comme un ensemble de protocoles légers (*lightweight protocoles*) : une couche *data link*, une couche *transport* orienté *connection less* basé sur la transmission de *datagrammes* et le tout couronné par un module de *Remote Procedure Call*.

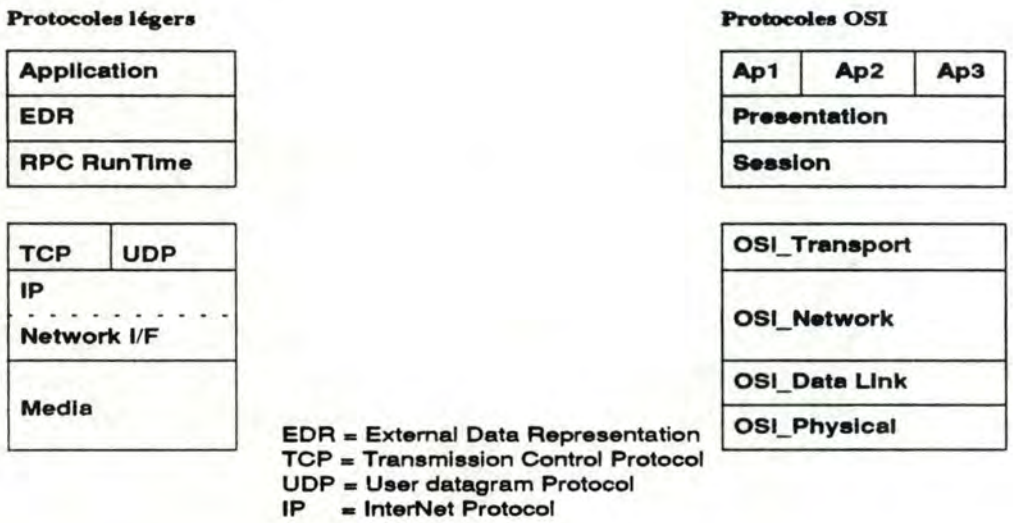

Schéma 2.3 : Protocoles légers

2.3.2 La communication client - serveur

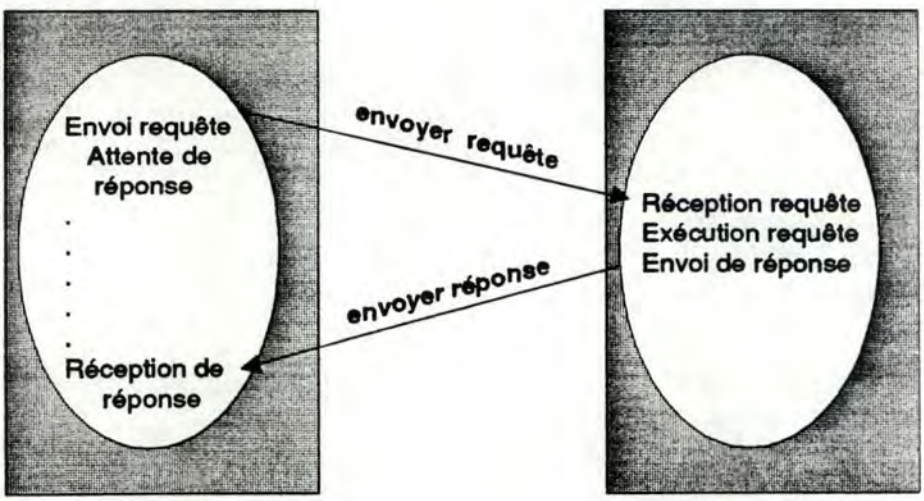

Schéma 2.4 : Communication client-serveur

18

La communication entre clients et serveurs se fait par l'envoi de messages. Le schéma 2.4 illustre cet échange.

Le client envoi un message de requête (*send request*). Le serveur reçoit cette requête (*receive request*), l'exécute (*execute request*) et renvoie éventuellement un message de réponse (*send reply*) à la requête. Les échanges peuvent évidemment être bien plus complexes lorsqu'il faut fragmenter des messages longs en unités plus petites ou lorsqu'il s'agit de garantir l'exécution d'une requête par l'envoi supplémentaire des messages de gestion du transfert tels que la confirmation ou l'indication de l'état du serveur.

Cette forme de communication nécessite donc au niveau transport l'existence de deux primitives :

. ***Send*** (*destination, data*)
. ***Receive*** (*source, buffer*)

Ces primitives peuvent être *bloquantes* ou *non-bloquantes*.

Un *Send* bloquant bloque le client jusqu'à la réception du message. Le *Receive* arrêtera l'exécution du serveur jusqu'à l'arrivée d'un message qui lui est destiné.

Les opérations non-bloquantes n'arrêtent point l'exécution des processus client/serveurs et communiquent avec l'application par mécanismes d'interruption ou d'envoi de signaux. Plus efficient en apparence, ce mode d'opération exige un gestion de buffers et d'interruptions plutôt complexe. Notons qu'il est possible de simuler sur un système bloquant le mode non-bloquant, au prix d'une couche additionnelle de gestion plutôt complexe.

L'existence des fonctions

. ***SendRequest*** (*destination, procId, dataIn*)
. ***GetRequest*** (*source, procId, dataIn*)
. ***SendReply*** (*source, dataOut*)

permet d'optimiser le protocole transport. Le client exécutant *SendRequest* est bloqué jusqu'à l'arrivée de la réponse à sa requête. Un serveur en attente dans *GetRequest* recevra la requête, l'identifiant de la procédure à exécuter (*procId*) et les paramètres de celle-ci (*dataIn*). Il exécutera la fonction en question, et renverra les résultats de l'exécution dans *SendReply*. La réception du message de réponse par le client le débloquera, et le client continue sa tâche. Trois appels seront donc suffisant pour une communication complète.

La communication par datagrammes étant non-fiable les erreurs de communication ne sont pas identifiées au niveau transport. Le software RPC devra donc s'occuper des erreurs suivantes :

- ***Messages perdus.*** Sans mécanisme de *timeout* la perte d'un message pourrait entraîner le blocage infini d'un processus.

- ***Messages dupliqués.*** Le software RPC filtrera ces messages par adjonction d'un *request identifier* et d'un *message identifier* univoques à chaque message.

- ***Messages hors séquence.*** Les clients étant bloqués jusqu'à la réception des réponses il leur sera facile de vérifier que la réponse arrivée correspond bien à la requête envoyée.

- ***Messages endommagés.*** Des mécanismes de *checksum* vont vérifier l'intégrité des paquets et en cas d'erreur en avertir les processus concernés.

3. Sécurité et protection des systèmes distribués

Les canaux de communication, les supports de stockage et les autres ressources d'un système distribué doivent être protégés contre l'accès accidentel ou volontaire non autorisé. Cette protection ne doit cependant pas entraver les possibilités de partage de ressources qui représentent l'atrait majeur de ces systèmes. Vu les lacunes de sécurité inhérentes aux systèmes distribués des mécanismes de protection ont été mis au point. Il s'agit en particulier de mécanismes de contrôle d'accès, d'identification et d'authentification des utilisateurs et de chiffrage.

3.1 Exposé du problème de sécurisation

Les systèmes distribués font l'objet d'attaques à leur sécurité. Ces attaques peuvent être de nature accidentelle ou volontaire. En ce qui concerne les attaques volontaires on relève surtout la lecture de fichiers confidentiels, l'exécution de processus dont l'usage est réservée aux utilisateurs privilégiés ainsi que l'utilisation illicite des canaux de communication. Les attaques volontaires sont effectués par des espions industriels, des fraudeurs professionnels et par des amateurs nommés "hackers". Ils essaient de passer au delà des mesures de protection en utilisant des moyens divers tels que la recherche illicite de mots de passe, le décodage manuel ou automatisé de fichiers chiffrés ou la mise en place de programmes virus qui simulent les fonctions du systèmes d'exploitation afin d'obtenir des informations secrètes.

Les caractéristiques spécifiques des systèmes distribués nécessitent des efforts de sécurisation supplémentaires. Parmi ces caractéristiques on relève l'existence de multiples points d'exécution des programmes et la fragilité des canaux de communication exposés.

3.1.1 Les multiples points d'exécution

Un processus particulier peut à priori s'exécuter en plusieurs endroits.

Cette possibilité possède l'avantage de séparer l'exécution propre des données à traiter. Il devient ainsi plus difficile de déterminer l'endroit exacte de l'attaque à effectuer.

Prenons l'exemple d'un espion industriel qui désire obtenir les plans d'un nouveau produit de l'entreprise X. Si ce plan se trouve centralisé en un endroit l'espion n'aura qu'à le localiser, casser les barrières de sécurité et de s'en emparer. Si ce plan se trouve dispersé sur une dizaine de machines répartis dans tout le pays, l'assemblage de toutes les pièces devient nettement plus difficile. Les mesures de sécurité peuvent varier d'un système à l'autre et les accès aux systèmes nécessitent éventuellement des mots de passe différents. Finalement le risque de découvrir la fraude augmente proportionnellement avec le nombre d'attaques perpétuées et le temps qui les sépare.

20

Si cependant certains noeuds du réseau se trouvent hors de contrôle cet avantage se transforme en un danger non mesurable. Quelles sont les politiques de protection de tel ou tel noeud. Les données considérées comme confidentielles chez nous peuvent être librement accessibles dans un autre endroit. La mise au point d'une politique de sécurité globale et des mesures pratiques qui l'accompagneront semble s'imposer ici.

3.1.2 Les canaux de communication exposés

Les noeuds d'un système distribué sont connectés entre eux par des canaux de communication. Il est techniquement faisable d'enregistrer le contenu de ceux-ci et d'y injecter des messages parasites. A la différence des systèmes centralisés où de telles attaques sont connues, les canaux de communication des systèmes distribués transportent non seulement les messages des applications mais aussi les messages du système lui-même.

Les appels de procédures lointaines (RPC) sont de tels messages système. Les messages de requête et de réponse peuvent contenir des données confidentielles et critiques. Un serveur peut être bloqué et surchargé suite à l'insertion frauduleuse de messages de requêtes bidons ou destructeurs.

La vulnérabilité est d'autant plus grande lors de l'échange de messages entre les composantes système comme les serveurs de fichiers. Les serveurs doivent pouvoir discriminer entre les messages système et les messages utilisateurs qui essaient de se faire passer comme tels.

Au contraire des grandes installations centralisés les connexion vitales du système se trouvent accessible de l'extérieur et toute attaque couronnée de succès peut avoir des conséquences désastreuses. Ceci explique les efforts faits dans le domaine du chiffrage des données privées et des procédés d'authentification des utilisateurs de l'installation.

3.1.3 Le nommage et la sécurité

La connaissance du nom d'un objet est déjà un pas vers son utilisation illicite. Si maintenant cette référence conduit directement vers l'objet, celui-ci devient d'autant plus vulnérable. Ceci est chose connue dans le milieu des services secrets depuis des siècles. Entre le nom public d'un agent secret et son existence physique se trouvent toute une série de pseudonymes tenus secrets.

Un procédé semblable trouve son utilisation dans la protection de systèmes distribués : entre les références *externes* connus par tous les utilisateurs et les objets à protéger, que ce soient des fichiers privés, des identificateurs d'hôtes, de ports, de transactions ou d'utilisateurs, se trouvent des noms *internes*. Le contrôle de la distribution et la protection de ces identificateurs internes est un aspect primordial de la protection de systèmes distribués.

21

3.1.4 Les systèmes ouverts (Open Systems)

Un système distribué devrait offrir un ensemble extensible de services. Cet ensemble comprend des service différents et parfois plusieurs versions d'un même service. Ces services sont rendus disponibles aux utilisateurs à travers des interfaces. Chaque application fera dès lors usage des interfaces aux services dont elle aura besoin au moment de son exécution.

Un système totalement ouvert permet la définition, l'extension, la modification libre et l'exploitation non restreinte des services du système et ceci par les utilisateurs eux-mêmes [LaSp]. Un système fermé offre un ensemble limité de services, l'extension et la modification de ces services étant réservés aux administrateurs et gérants du système.

Une certaine ouverture des systèmes distribués est inévitable. L'objectif de protection des domaines d'objets appartenant aux divers utilisateurs s'oppose clairement à celui du partage des ressources et de la communication libre. La messagerie électronique en UNIX [Unix] est un exemple classique : Elle permet à un utilisateur, origine d'un message, d'écrire dans une boite à lettre, élément du domaine de protection du destinataire. La protection stricte des espaces disques alloués à un utilisateur s'oppose aussi au partage de fichiers.

Or les conséquences de cette ouverture peuvent faciliter les attaques à la sécurité de l'installation entière. Ces attaques prennent souvent la forme de ***virus*** ou de ***chevaux de Troie*** (*troyan horses*).

Un *virus* est un programme qui se multiplie et se propage dans le réseau en consommant ses ressources et en circonvenant les mécanismes de protection disponibles. Il est composé de deux parties. L'une s'occupe de sa reproduction et l'autre de la mission à exécuter. Cette mission varie d'un virus à l'autre. Elle peut aller de l'affichage d'un simple message d'avertissement à la consultation, de la destruction de fichiers privés ou de fichiers systèmes jusqu'à l'arrêt total du système. Un virus peut même trouver des utilisations bénéfiques [ShHu].

Le *cheval de Troie* prend son nom de la célèbre bataille de Troie. Il s'agit d'un programme en apparence normal et utile et qui pour cette raison se trouve vite accepté et utilisé par tous les utilisateurs. Sans perception des utilisateurs et souvent en parallèle avec sa tâche apparente, le cheval de Troie exécute sa vraie mission. Avec la priorité et les privilèges obtenus de l'utilisateur victime il accède à des données tenus secrets et les transmet à des personnes non autorisées. Un exemple classique de cheval de Troie est celui où un programme, une fois exécuté par la victime, laisse dans l'espace disque du fraudeur un programme qui prend les privilèges de la victime lors de son exécution. L'utilisateur qui dispose alors de ce programme possède les mêmes privilèges que sa victime.

22

Ces deux mécanismes se trouvent parfois combinés. Prenons l'exemple d'un virus appelé "*Christmas tree virus*". Il se propageait à travers le réseau par le biais de la messagerie électronique. Le message arrivé dans la boite de lettre invitait l'utilisateur à exécuter un petit programme qui lui souhaitait un joyeux Noël. Derrière cette exécution visible le virus consultait les listes de distribution des messages électroniques et envoyait une copie de soi même à chacune des adresses y figurant. Le virus a ainsi accompli son travail : il s'est répliqué et a effectué sa mission qui dans ce cas était l'affichage du message "Joyeux Noël". On retrouve aussi les caractéristiques du cheval de Troie puisque derrière une apparence bénigne des violations de confidentialité ont été effectuées.

La tâche des virus et des chevaux de Troie est facilitée par le fait que beaucoup de systèmes confèrent par défaut au programme exécuté les privilèges de l'utilisateur qui les lance. L'origine du programme n'est pas prise en compte. Si le système laissait aux exécutables les privilèges de leur créateur jusqu'à l'adoption définitive par l'utilisateur concerné l'on pourrait bien prévenir les attaques du genre "Christmas tree virus".

3.2 Le mécanismes de protection

3.2.1 Domaines de protection et "capabilities"

Un processus accède à plusieurs ressources. L'ensemble des objets utilisés par un processus s'appelle *domaine de protection*. [Lampson][GrDe] Les domaines de protection peuvent être isolés ou se recouper avec d'autres, ce qui est le cas pour les ressources partagées. L'appartenance d'un processus à un domaine spécifique est détermine par l'*authentification*.

Il est du ressort du système d'exploitation de protéger les domaines de protection contre les interférences par d'autres processus. Cette protection se fait ou bien par l'usage de hardware spécifique, dans le cas d'accès à la mémoire ou de l'utilisation du réseau, ou bien par le mécanisme des *capabilities*.

Une *capability* est une référence ou un nom qui confère à son titulaire le droit d'utilisation d'une ressource hardware ou l'accès à des données stockées sur mémoire de masse. Elle permet d'identifier les objets comme les fichiers, les directories et les ports de communication partagés entre processus ayant des droit d'accès différents. Les droits et privilèges offerts par la *capability* doivent être cachés ou protégés de toute manipulation par les processus utilisateurs.

Elle se compose :

1. d'un identificateur unique pour l'objet auquel on fait référence
2. de la spécification des droits d'accès
3. d'une partie à valeur aléatoire

Les deux dernières parties se trouvent parfois mélangées.

23

Les *capabilities* sont distribuées et gérées par un *serveur de capabilities* du software système. Celui-ci fournit les droits d'accès à ses différents clients. En principe le processus créateur d'un nouvel objet détient initialement tous les droits d'accès sur celui-ci. Si l'objet est privé, s'il s'agit d'un fichier confidentiel ou d'une directory à protéger, le système refuse tout accès aux autres processus. Si l'objet est partagé le système alloue les droits qui conviennent aux différentes catégories d'utilisateurs et de processus.

Evidemment le propriétaire peut modifier les droits d'accès de son objet. Il n'aura qu'à révoquer sa *capability* pour que le serveur lui fournisse une nouvelle. Le champ aléatoire de la nouvelle *capability* sera différent de l'ancienne et tous les clients devront redemander l'obtention de celle-ci au serveur.

Les *capabilities* forment le lien entre la référence, le nom d'un objet et l'objet lui-même. Cette association doit être tenue secrète et accessible uniquement aux processus autorisés. L'*authentification*, mécanisme de détermination de l'utilisateur pour le compte duquel un processus tourne, garantit cet accès sélectif aux informations internes des *capabilities*.

Pour éviter la modification ou la fraude de *capabilities* celles-ci doivent être protégées. Les protection hardware, par segments de mémoire protégée, ne sont pas applicables aux systèmes distribués et se trouvent remplacés par des techniques logicielles. Sans cette protection les processus utilisateur pourraient créer eux-mêmes ou modifier les *capabilities* dont ils disposent afin d'obtenir un accès non-autorisé aux ressources ou fichiers du système.

L'ensemble des domaines de protection d'un système distribué peut être décrit sous forme de la matrice d'accès :

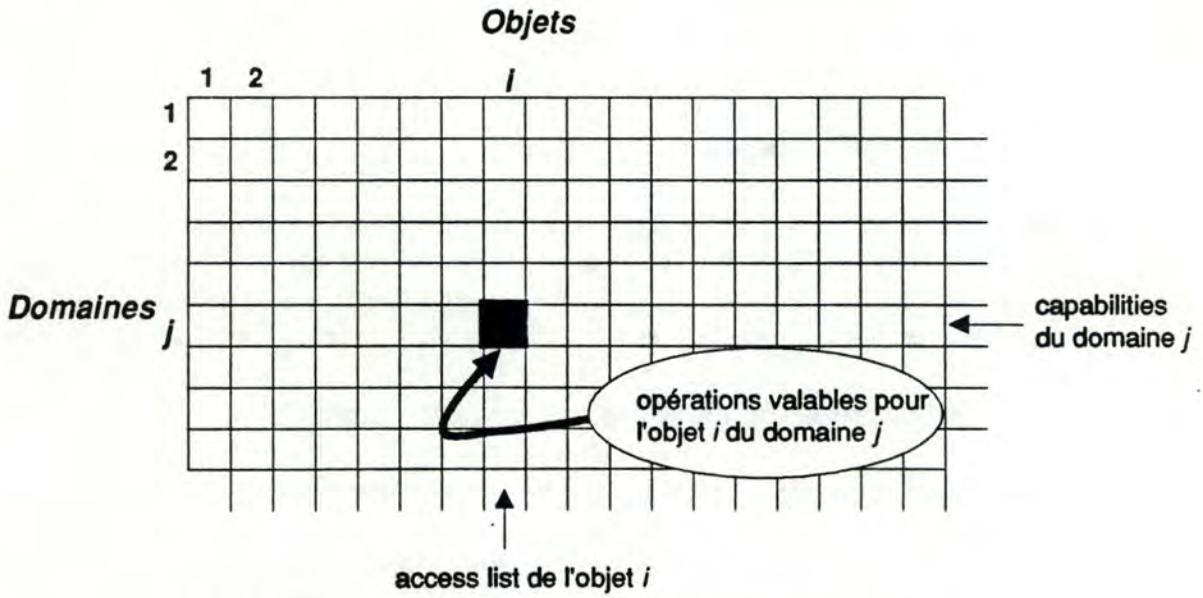

Schéma 3.1 : Représentation matricielle des domaines de protection

24

La *matrice d'accès* relie les différents objets du système, en abscisse, aux domaines de protection, en ordonnée. Dans cette matrice une colonne représente une *liste d'accès* (*access list*) d'un objet, identifiant tous les domaines qui y ont accès ainsi que la nature exacte de leurs droits : lecture,écriture et modification. Les *ensembles de capabilities* (*capabilities set*) sont représentés par les lignes de la matrice et énumèrent toutes les *capabilities* détenues par un domaine de protection. L'appartenance à un domaine de protection est gérée par le mécanisme d'authentification.

Il existe plusieurs stratégies de protection d'objets gérés par un serveur lointain, sachant qu'il dispose de sa propre matrice d'accès.

1. le serveur vérifie si la *capability* fournie par le client lui donne droit d'exécuter l'opération demandée

2. le serveur *authentifie* le client et parcourt la liste d'accès pour l'objet concerné par la requête.

Puisque l'authentification à chaque requête peut être plutôt lourde dans le cas, par exemple, de serveurs de fichiers et que le contrôle des *capabilities* minimise le stockage des informations d'état sur les différents clients les administrateurs optent souvent pour la deuxième stratégie. Une solution souvent adoptée consiste en une authentification unique lors de l'entrée de l'utilisateur dans le système. Toutes le protections ultérieures consistent alors dans la vérification des *capabilities*.

3. Une stratégie hybride se trouve implémentée dans les serveurs de fichiers transactionnels qui font l'authentification uniquement à l'ouverture de la transaction. Suite à celle-ci le client obtient une *capability de transaction* temporaire, qu'il faut présenter à chaque opération liée à la transaction en cours.

3.2.2 L'authentification

Nous venons de voir que les processus des utilisateurs tournent dans des domaines de protection qui définissent les types d'opérations qui peuvent être effectués sur les objets qui en font partie. La détermination de l'appartenance d'un processus à un domaine est la tâche de l'authentification.

A) L'authentification dans le systèmes centralisés

Le système garde un fichier de passwords qui associe au nom de chaque utilisateur un mot de passe chiffré. Lors du *login* l'utilisateur fournit son nom ainsi que son mot de passe. Si les informations fournies concordent avec ceux du fichier password l'utilisateur est authentifié et son domaine de protection est crée. Il disposera dès lors d'un ensemble de droits et de privilèges gérés et vérifiées par le *Kernel* du système d'exploitation.

25

B) L'authentification dans le systèmes distribués intégrés

La situation est semblable à celle des systèmes centralisés [LCS 1.]. Un fichier, répliqué sur toutes les machines, contient les informations d'identification des utilisateurs : le nom, l'identificateur interne (*user identifier*, UID) le groupe auquel on appartient (*group identifier*, GID) et un mot de passe chiffré. L'authentification et les contrôles d'accès sont effectués par le *Kernel*, ce qui garantit la sécurité du système et la protection des données.

C) L'authentification dans les systèmes ouverts basés sur l'architecture client-serveur

Ici la protection devient plus compliquée puisque :

- les utilisateurs travaillent sur des stations de travail. Il n'y aucune raison pour qu'un serveur fasse confiance aux informations d'authentification en provenance de celles-ci

- les opérations des clients, sujets au contrôle d'accès, sont demandés par les clients mais effectués par des serveurs sur de machines lointaines.

Chaque serveur gère des ressources locales et dispose de sa propre matrice d'accès. Il doit vérifier la validité de chaque requête fourni par ses clients. Une solution serait de demander en argument à chaque requête le domaine de protection du client. Or le client peut se trouver sur une autre machine entièrement hors contrôle du serveur et fournir des informations frauduleuses et incorrectes.

Une solution de protection plus solide pour ces systèmes est celle d'un *serveur d'authentification des utilisateurs* (*user authentication server*) [LCS 4.]. Lors du *login*, l'utilisateur s'adresse au serveur d'authentification. Le *serveur des noms actifs* (*active name server*) lui fournit un *jeton d'authentification* (*authentication token*) difficile à forger.

Les clients fournissent alors comme argument à toute requête ce jeton ainsi que leur identification. Le serveur cible s'adresse ensuite au *serveur des noms actifs* pour valider les informations fournis.

Celui-ci dispose d'une table qui reprend toutes les informations concernant les jetons attribués :

1. l'identificateur permanent de l'utilisateur
2. le jeton d'authentification
3. l'identificateur de l'autorité créateur du jeton
4. un jeton de contrôle

26

Il dispose des primitives de service suivantes :

GetToken(*userId,authority,oldToken*)
-> *newToken,ControlToken*

Lors du login le serveur d'authentification utilise cette fonction pour obtenir le jeton d'authentification accompagné de son jeton de contrôle et le passe à l'utilisateur. La primitive entraîne la création d'un nouvel enregistrement dans la table d'authentification.

Verify(*authToken,userId,authority*) -> *tokenOk*

Cette primitive, à disposition des serveurs d'application, vérifie si le token (*authToken*) correspond bien à l'utilisateur demandeur (*userId*) et s'il a été émis par l'autorité convenable (*authority*). Le résultat est un booléen dont la valeur est *true* si les informations sont validées et à *false* sinon.

InitializeTable

Cette commande privilégiée permet d'initialiser les tables du serveur des noms actifs sans token d'autorisation. Ce service n'est évidemment disponible qu'au serveur d'authentification.

3.2.3 La protection des ports

Les *ports* ou *sockets* sont les extrémités abstraites de la communication entre deux processus. Le *Remote Procedure Call*, extension de l'appel procédural classique, utilise la couche transport du logiciel de télécommunications pour envoyer et recevoir à des adresses *socket*,les messages échangés entre le client et le serveur.

Les *ports*, autant que les canaux de communication, sont exposés aux attaques extérieures: la lecture passive d'informations qui y arrivent, l'envoi actif de messages incohérents ou destructeurs, et la modification de messages en cours de transmission. Le système essaie de se protéger contre la plupart de ces attaques en mettant en oeuvre des mécanismes de chiffrage, ce qui empêche la lecture illicite de messages en transit, ainsi que des techniques de détection de messages endommagés, techniques de *checksum*. Remarquons que sans protection suffisante des *sockets* un saboteur pourra toujours tenter de ralentir les performances du système en saturant les serveurs critiques en leur envoyant des masses de messages bidons. Il est facile de bloquer un serveur de fichiers en lui envoyant constamment des requêtes de lecture de grands blocs sur disque.

Une protection efficace des *ports* est inévitable dans un environnement sécurisé. Elle est souvent réalisée par le mécanisme des *capabilities* qu'on applique aux *sockets*. Par la présentation de la *capability* son détenteur obtient des droit d'accès en lecture et en écriture sur un *socket*.

Or, dans les architectures client-serveur, l'on doit distinguer entre les acteurs et leurs droits respectifs. Le droit de lire sur le port d'un serveur les messages de requête en arrivée devrait être limité à ce serveur spécifique. Ce même port peut toutefois être accessible en

27

écriture à une multitude de clients. De même le droit de lecture sur le port d'un client doit être réservé à ce client. Le client donne au serveur, pour que celui-ci puisse renvoyer sa réponse, un droit d'écriture sur le *port* en question. Pour réaliser cette protection différenciée, le système associe des droits d'écriture et de lecture aux *capabilities*.

On aura alors besoin d'un agent qui s'occupe de distribuer les droits respectifs et de vérifier à ce que les différents processus les respectent. Différentes implémentations de protection de *ports* ont vu le jour :

a) Dans le système Mach toute communication entre machines passe par un service réseau avec un serveur réseau sur toute machine [Sansom]. Seul un serveur aura le droit d'écouter sur un port et les jetons d'autorisation sont composés :

1. d'un identificateur publique unique
2. d'un identifiant secret qui reprend les droits d'écriture de façon à ce que tout détenteur du jeton ait des droits d'écriture
3. de l'adresse de l'ordinateur ayant le droit de lecture
4. de l'adresse de l'ordinateur propriétaire du port

Le serveur réseau créateur d'un port possède initialement les droit de lecture sur celui-ci. Pour permettre la communication il passera aux client qui le demandent un jeton d'autorisation d'écriture ainsi que son jeton d'authenticité. Ce dernier jeton permettra aux client de vérifier l'authenticité d'un serveur qui se présente comme lecteur du port en question.

Le serveur réseau peut passer ses droits de lecture à un autre serveur. Suite à la transmission au nouveau serveur de sa capability de lecture ainsi que de son jeton d'authenticité, l'ancien serveur n'aura plus le droit d'écouter les messages en destination du port. Sa tâche se limitera alors de transmettre aux demandeurs l'adresse du nouveau serveur à contacter. De nouveau les clients peuvent utiliser le jeton d'authenticité pour démasquer des serveurs illicites.

b) Dans le système Amoeba [MuTa] les clients et serveurs disposent d'un interface réseau sécurisé. Un port est protégé par des *capabilities* d'envoi et de réception de messages, appelés respectivement *put-port* et *get-port*. Un socket se compose de cette paire de jetons (*put-port,get-port*). Le message envoyé au *put-port* n'est reçu que par le détenteur du *get-port* qui y correspond. Tandis que le jeton *put-port* est connu des autres processus le *get-port* reste le secret de son propriétaire.

Le *jeton put-port* ne comporte pas l'adresse du destinataire. Lorsqu'un processus envoit un message à un *put-port*, le software de transmission doit s'occuper de trouver le serveur qui dispose de la paire (*put-port,get-port*) correcte. Ceci peut se faire par l'envoi d'un message broadcast qui enquête sur l'adresse du processus propriétaire du *put-port* en question.

28

Pour que le message envoyé à un *put-port* puisse être lu par le détenteur du *get-port* le système doit disposer d'un mécanisme de mapping faisant l'association entre ces deux *capabilities*. Il s'agit d'une fonction qui s'exprime comme :

put-port = F(*get-port*), où F est la fonction de mapping.

Lorsqu'un serveur non valable répond à une enquête broadcast de localisation il ne pourra point recevoir des messages de client puisqu'il ne disposera pas d'une paire (*put-port,get-port*) valide. Pour éviter que l'on puisse dériver le *get-port* à partir du *put-port* la fonction de mapping doit être telle qu'il sera difficile d'établir son inverse. Pour garantir la sécurité du système Amoeba l'on pose des "*boîtes fonctionnelles*" (*function boxes, F-Boxes*) entre les ordinateurs et le réseau. Celles-ci peuvent être implémentées en hardware ou en software et leur travail est d'appliquer une fonction non- ou difficilement inversible (*one way functions*) aux *capabilities* des *ports*.

Schéma 3.2 : La protection des ports en Mach

Le client, lorsqu'il fait une appel procédural remote, envoit son message de requête au *put-port* du serveur, S_p tandis que celui-ci est occupé à écouter sur son *get-port* S_g. Ce message comprend entre autres le *get-port* du client, C_g sur lequel il reçoit les réponses du serveur ainsi les arguments en entrée (*data*). Les *boîtes fonctionnelles* entrent alors en jeu en appliquant la fonction de mapping aux *get-ports* pour obtenir les *put-ports* qui y sont associés et vice-versa. Le message transmis n'arrive qu'a destination si les capabilities envoyées en argument par le client sont correctes, c'est à dire si le *put-port* envoyé en argument, S_p du serveur correspond bien au *put-port* calculé à partir du *get-port*, S_g, du serveur. Le même raisonnement est valable pour le renvoi des réponses du serveur au client.

3.2.4 Le chiffrage des données

L'idée de chiffrage est de transformer les données à protéger de façon à ce que seul le destinataire prévu, qui dispose d'un moyen de les déchiffrer, puisse les lire. Cette protection est efficace contre les attaques par processus 'imposteurs' ainsi que les lectures directes des canaux de communication [Kahn][Denning].

a) Les règles de transformation et les clés

Le message à transmettre est chiffré par son origine en appliquant une *règle de transformation* du texte *clair* vers le texte *chiffré*. Seul celui qui dispose de l'inverse de cette règle serait en mesure de reconstruire le texte clair d'origine en l'appliquant au texte chiffré. Ces règles de transformation consistent en des transpositions et remplacements de lettres. Notons que du moment qu'une règle est découverte elle n'a plus d'utilité et l'on doit en mettre au point une nouvelle.

29

Afin d'éviter ce cercle vicieux, les règles de transformations sont devenus plus sophistiquées. Une règle est constituée de deux parties : une *fonction* et une *clé*. Chiffrer un texte consiste alors en appliquant la fonction de transformation au texte clair en tenant compte de la clé. Celui qui dispose de la clé pourra alors appliquer la fonction inverse de déchiffrage au texte chiffré et reconstruire l'information utile du message. Au contraire de la clé, qui doit rester strictement confidentielle et secrète, les spécifications des fonctions de transformation sont souvent publiques. Le schéma qui suit illustre ce mécanisme. L'origine (sender) chiffre le texte clair (p) avec une clé (K). Le texte chiffré (c) passe à travers le réseau et arrivé au destinataire (receiver) qui lui le déchiffre avec la même clé (K) pour reconstituer le texte clair envoyé (p).

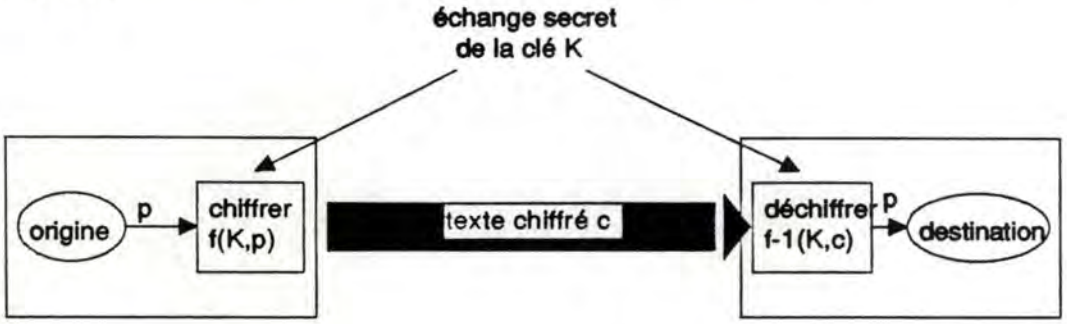

Schéma 3.3 : Chiffrage à clé secrète

Le but de tout "*casseur de code*" est alors de découvrir la clé de transformation. Cela peut se faire en comparant des textes clairs et leur équivalents chiffrés pour y trouver des similarités qui pourraient aider à retrouver la clé. Cette tâche est cependant rendue très difficile par l'usage de clés longues ainsi que de fonctions d'encyptage non linéaires. Le standard mis au point et adopté par les instances gouvernementales des Etats Unis, le DES (*Data Encryption Standard*) comporte un algorithme compliqué de 19 étapes et une clé de 56 octets. Le standard [NBS_DES] se trouve, pour des raisons de sécurité et de performance, souvent implémenté sous forme de hardware VLSI et ceci sur un circuit unique.

b) La distribution de clés

Nous avons vu précédemment que l'origine et la cible d'un message doivent en connaître la clé de chiffrage. La transmission des clés par voies classiques, par téléphone ou par courrier interne n'est pas très commode. C'est la raison pour laquelle on a introduit l'idée d'*un serveur de distribution de clés* (*key distribution server*) qui fournit des clés secrètes aux instances désireuses de communiquer en sécurité. [NeSch]

La communication entre les clients et le *serveur de clés* doit être protégée. Ce serveur dispose d'une table contenant la clé privée de tout utilisateur du système et ne communiquera alors avec ses clients qu'au moyen de celle-ci. La *clé privée* permet l'authentification des utilisateurs ainsi que le chiffrage des messages échangés entre les processus utilisateur et le serveur des clés. Si un message vers le serveur est chiffré avec la clé privée de l'utilisateur et que le serveur des clés peut le déchiffrer par la clé privée qu'il trouve dans ses tables la preuve de l'authenticité du client est fournie puisque les autres processus ne sont pas censés connaître cette clé. Inversément le serveur chiffre les messages en destination de ses clients. Si un client ne peut déchiffrer le message il est fort probable qu'un processus frauduleux ait tenté de se faire passer comme serveur des clés.

30

Le serveur des clés génère aussi des *clés secrètes* pour sécuriser la communication entre processus. Si un de ses clients veut échanger des informations avec un autre, il s'adresse au serveur des clés. Celui-ci lui répond par l'envoi d'un message contenant la clé secrète, et cela sous deux formes : l'une compréhensible par le premier client et l'autre chiffrée par la clé privée du deuxième client. Le client ayant demandé la clé sortira du message la clé chiffrée pour l'autre processus pour la lui envoyer. A ce stade les deux processus sont en possession de la clé secrète et peuvent communiquer.

Le protocole se trouve illustré dans le schéma 3.4 où la *f(k,i)* dénote la fonction de chiffrage *f* appliquée sur l'information *i* avec la clé *k*.

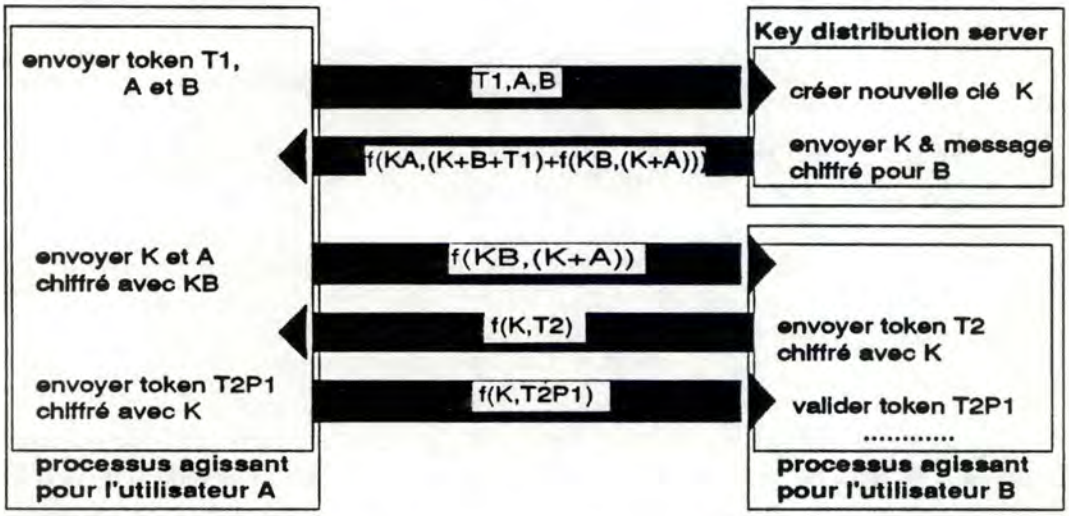

Schéma 3.4 : Le serveur de distribution de clés

Un processus lancé par l'utilisateur A désire communiquer avec un autre processus qui tourne sous le compte de l'utilisateur B. Le serveur des clés dispose des clés privées de A (*KA*) et de B (*KB*). Le processus A envoie ensuite un message de requête au serveur des clés contenant un jeton provisoire de contrôle de concordance entre la requête et la réponse obtenue (*T1*), ainsi que l'identification des utilisateurs impliqués, *A* et *B*. Le serveur vérifie l'authenticité de la requête et compose un message contenant

1. *K*, *A* et *T1*
2. (*K* et *A*) chiffrés avec la clé privée de B

où *K* constitue la clé de communication entre *A* et *B*. Ce message est chiffré par la clé de *A* et renvoyé au demandeur. Celui-ci, après avoir déchiffré la réponse obtenue,en enverra la deuxième partie au processus de *B*. Après déchiffrement *B* dispose de la clé nécessaire. Pour s'assurer que le message reçu n'est pas le résultat d'un message dupliqué par erreur dans le réseau mais qu'il provient effectivement de *A*, *B* renvoit alors un autre jeton de contrôle provisoire *T2*, chiffré cette fois-ci avec la clé *K*. Si le message qui contenait *K* provient vraiment de *A*, ce processus sera capable de le déchiffrer. Il appliquera une fonction connue à *T2* et renverra le résultat, de nouveau chiffré à *B*. Après une vérification par le processus *B* l'on peut considérer que les deux processus sont en communication protégée.

31

c) La signature digitale

Dans un document écrit la signature d'une personne l'authentifie et vaut comme preuve de consentement de cette personne. Dans les systèmes bureautiques actuels des documents sont constamment transmis d'un utilisateur à un autre. Il serait intéressant de disposer d'un mécanisme de *signature électronique* (*digital signature*) dans le cadre de systèmes distribués [NeSch]. Mais comment faire la preuve de l'authenticité d'une telle signature ? Comment éviter que le signataire éventuel révoque sa validité ?

Dans ce cas aussi le serveur de distribution de clés peut nous être utile. Le *service de signature digitale* garantit au destinataire du document la validité légale du document envoyé. Supposons qu'un processus de l'utilisateur A envoit un document à un autre processus, qui tourne dans le domaine de protection de B.

1. Le processus A qui désire ajouter une signature digitale au document concerné va chiffrer le texte clair avec sa clé privée et le transmettre au serveur des clés. Celui-ci le déchiffre et ajoute un certificat daté et signé au document. Ce certificat, chiffré par une clé secrète du serveur des clés, résulte du compostage du nom de l'utilisateur A, du document signé et de l'heure et de la date de validation. A recevra une copie de ce certificat.

2. A envoit une copie du certificat reçu vers B. Avant de l'envoyer au serveur des clés B en fait une copie sur disque. Le serveur des clés reçoit le certificat qu'il déchiffre avec sa propre clé privée. Le document est enfin envoyé, chiffrée par sa clé privée, à B.

3. Le processus de B reçoit le document chiffré et le déchiffre. B dispose maintenant du document clair et d'une copie du certificat d'authenticité fourni par le serveur des clés.

Il est à remarquer ici que les processus A et B font confiance au serveur des clés. B est rassuré que le document est signé puisqu'il en a obtenu un certificat. Il sera très difficile à A de révoquer sa signature car B possède un certificat validable par le serveur des clés. Ce certificat n'a pu être forgé par B puisque celui-ci n'est pas en connaissance de la clé secrète du serveur.

d) Le chiffrage avec clé publique

Le *chiffrage par clé publique* évite l'échange de clés à travers le réseau. Chaque récepteur potentiel engendre deux clés, une *clé d'envoi*, de chiffrage (K_e) et une *clé de réception*, de déchiffrage (K_d). La clé d'envoi K_e est publique et peut être utilisé par tout processus désireux de communiquer tandis que la clé de réception K_d reste secrète. L'association entre ces deux clés est gérée par une fonction de mapping. Cette fonction doit être telle que la clé secrète de réception ne soit pas ou du moins difficilement dérivable de la clé d'envoi.

32

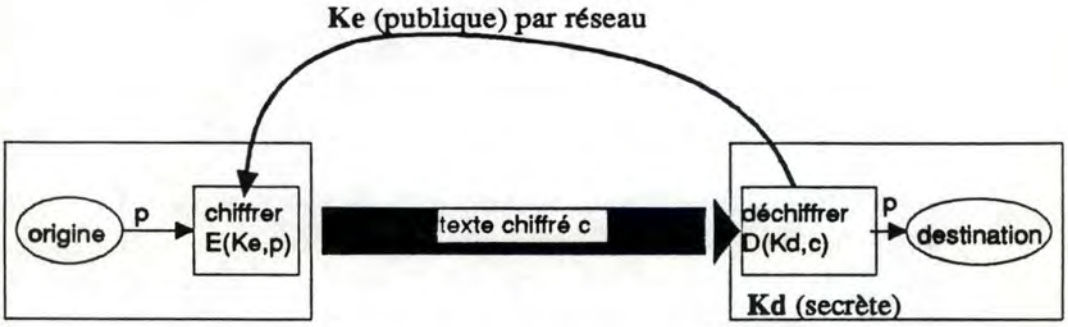

Schéma 3.5 : Chiffrage à clé publique

L'idée se base sur l'existence de deux fonctions D et E appliqués à deux clés différentes K_e et K_d Si A désire obtenir des informations de B, il génère ses clés et fait passer la clé d'envoi à B. B applique alors la fonction E au texte clair pour le chiffrer avec la clé d'envoi. Le texte chiffré est transmis à A qui utilise la fonction D pour déchiffrer le texte avec la clé de réception. Seul A est en mesure de déchiffrer le message comme il est le seul à connaître la clé de réception.

La fonction de mapping entre clé d'envoi et clé de réception doit être "one way", c'est à dire qu'elle ne devrait pas posséder de fonction inverse ou que celle-ci soit difficile à déterminer. Rivest, Shamir et AdelMan (RSA) [RiShAd] ont mis au point un algorithme de chiffrage basé sur la difficulté de trouver les facteurs des grands nombres.

Il est à remarquer que même si ce modèle de chiffrage ne se base pas sur l'existence d'une clé de chiffrage tenue secrète l'utilisation d'un serveur de distribution de clés peut s'avérer utile. Sa tâche sera notamment de s'occuper de la distribution des clés publiques aux demandeurs éventuels et d'authentifier les processus serveurs qui y publient leurs clés.

33

e) A quel niveau effectuer le chiffrage ?

Nous avons vu que le software réseau d'un système distribué est constitué de couches, de modules indépendants qui communiquent de façon hiérarchique. L'information à transmettre d'un processus à un autre peut donc être chiffrée à un ou plusieurs de ces niveaux allant du niveau application au niveau physique en ce qui concerne le modèle OSI.

Les concepteurs du modèle OSI ont prévu la couche présentation qui doit entre autres s'occuper du chiffrage/déchiffrage de messages en provenance des applications. Pourquoi alors ne pas faire le chiffrage au niveau le plus élevé, c'est à dire par l'application elle-même. Ceci aura l'avantage de disposer de messages chiffrés depuis leur envoi par l'application. Le danger de lecture illicite de messages tant qu'ils se trouvent dans les files d'attente du système se trouve ainsi éliminé. Or cela n'empêche pas des analyses de trafic illicites des données transmises puisque les headers et trailers ajoutés aux paquets du niveau 7 par les niveaux inférieurs restent en texte clair.

Birrell et Nelson, dans leurs documents sur le Remote Procedure Call, ont prévu un mécanisme de chiffrage des messages de requête et de réponse échangés entre les clients et serveurs d'un système distribué [BiNe]. Celui-ci se base sur le standard DES et sur l'utilisation d'un serveur de clés : *GrapeVine*.

Le chiffrage aux niveaux inférieurs cache bien les adresses des niveaux supérieurs et empêche la détermination précise des adresses d'origine et de destination des paquets. Le problème de cette démarche réside dans le fait que les informations envoyés par les applications se trouvent en texte clair dans le système jusqu'à leur arrivée aux niveaux data-link ou même physique. Jusqu'à leur chiffrage ils peuvent devenir la proie de virus et de chevaux de Troie.

Selon les besoins en sécurité il peut s'avérer utile d'effectuer un chiffrage à plusieurs niveaux, en prenant en compte le surplus de ressources que cela entraîne.

34

Chapitre II. Remote Procedure Call - Théories

1. L'idée de Remote Procedure Call

Les architectures de communication actuelles se voient confrontés à une masse croissante de protocoles divergents et parfois très complexes. Le développement du modèle OSI n'a guère changé cette situation. Il ne fait que fournir une base de référence. Chaque catégorie d'application aura ainsi besoin de son propre protocole de transmission.

Les désavantages de cette solution sont multiples.

1) **La diversité et la complexité des protocoles de communication** fait augmenter le prix de l'implémentation et de la maintenance des applications distribuées et réduit nettement leur interopérabilité.

2) **L'ampleur de certaines implémentations** et la dynamique des interactions de certains protocoles s'opposent aux objectifs de fiabilité et de validation dont un environnement sécurisé est dépendant.

3) **Le développement incontrôlé de protocoles** rend la mise au point de hardware d'optimisation de protocoles quasi impossible.

L'approche ***Remote Procedure Call*** (RPC), qui trouve déjà son utilisation dans de nombreux systèmes distribués, permet de consolider les diversités des protocoles d'application dans le modèle abstrait de l'appel procédural. Cette architecture se trouve implémentée par un protocole unique : le protocole RPC.

Les serveurs mettent leurs fonctionalités à disposition des clients sous forme de l'interface d'un module. Les appels aux fonctions lointaines faisant partie de ce module seront automatiquement traduits en appels vers des *stubs*, sortes de procédures locales de remplacement, qui eux utilisent les services de communication des couches inférieures pour invoquer le service demandé.

2. Objectifs

- **Faciliter le développement d'applications distribuées.** Le programmeur n'aura plus à se soucier des aspects de communication inter-machines. Puisque le RPC est basé sur l'appel procédural il pourra utiliser la méthodologie de décomposition modulaire bien connue pour le développement d'applications centralisées. Il lui restera cependant à gérer la synchronisation des tâches, les failles indépendantes de certaines composantes et l'exécution de processus situés dans des environnements indépendants.

- **Performance et efficacité des RPC.** Un service RPC lent et inefficace peut décourager son utilisation. Les développeurs vont ou bien éviter la distribution de leurs applications ou bien écrire leurs propres protocoles d'application, incompatibles, difficiles à utiliser et peut-être même mal ou

35

peu documentés. Afin d'attirer les développeurs à l'utilisation du RPC il est désirable d'optimiser les logiciels de communication. L'utilisation de *threads* (processus légers) et de modes de communication performants (communication par *datagrammes*) sont des pas dans la bonne direction.

- **Sémantique puissante des RPC.** La sémantique des RPC doit être la plus proche des appels procéduraux locaux. L'utilisation des RPC devra être le plus transparent possible à l'utilisateur. Les passages de paramètres par référence et des structures complexes imbriquées et les failles du réseau sont quelques uns des points à prendre en considération.

- **Les aspects de sécurité.** Les RPC ont des besoins de sécurisation supérieurs aux appels locaux. En effet on n'a pas le même type de contrôle des ordinateurs lointains que des machines locales. Comment savoir qu'un client s'adresse au bon serveur, qu'un serveur ne fournit pas des informations confidentielles à un client non autorisé ?

3. Notions de base

3.1 L'appel procédural

L'appel d'une procédure équivaut à passer le contrôle à une séquence d'instructions qui constitue le corps de la procédure. Les environnements des parties concernés sont éventuellement modifiés par le passage de paramètres. Ceux-ci peuvent être de plusieurs sortes : en entrée, en sortie, par valeur ou par référence. Le passage par valeur ou par référence entraîne respectivement le transfert d'une copie des valeurs ou de l'adresse de l'objet concerné vers l'environnement de la procédure appelée. Une fois la séquence d'instructions du corps achevée, le contrôle repasse au programme appelant. Les paramètres en sortie ou en référence vont évidemment modifier son environnement.

Le diagramme des environnements pour un exemple est représenté au schéma 3.2 .

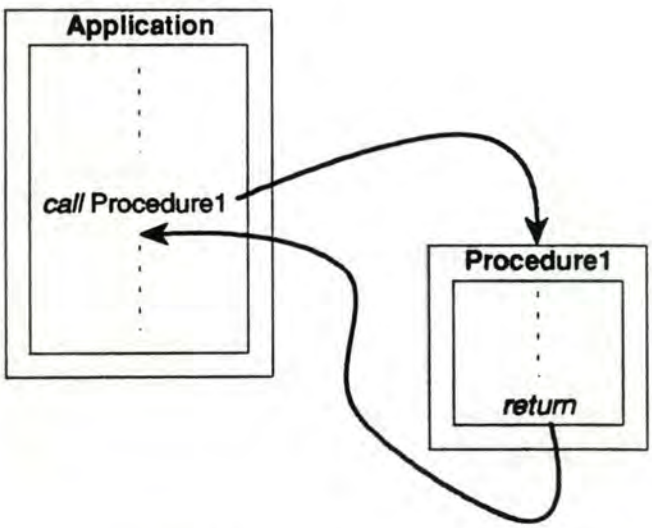

Schéma 3.1 : Déroulement d'un appel procédural

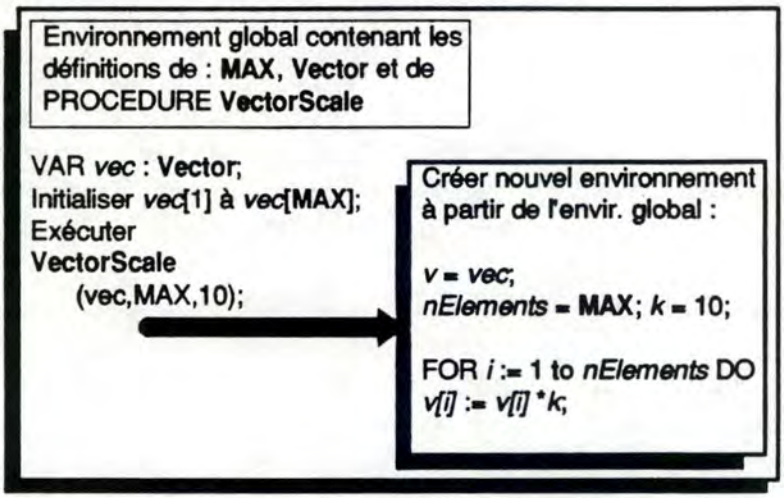

Schéma 3.2 : Diagramme des environnements pour l'appel procédural local à VectorScale

3.2 Le concept de module

Un *module* est un regroupement de fonctions ayant trait à l'exécution d'une même tâche. Il comprend un certain nombre de *fonctions* mis à la disposition des applications à travers une *interface*. Celle-ci décrit les fonctions, ainsi que le nombre et le type des paramètres en entrée et en sortie. Elle est rendu accessible aux applications par l'opération d'*exportation*. Toute application désireuse d'accéder aux fonctions du module va réaliser une *importation* de son interface. Le module constitue une unité indépendante et peut ainsi être encapsulé par un serveur lointain.

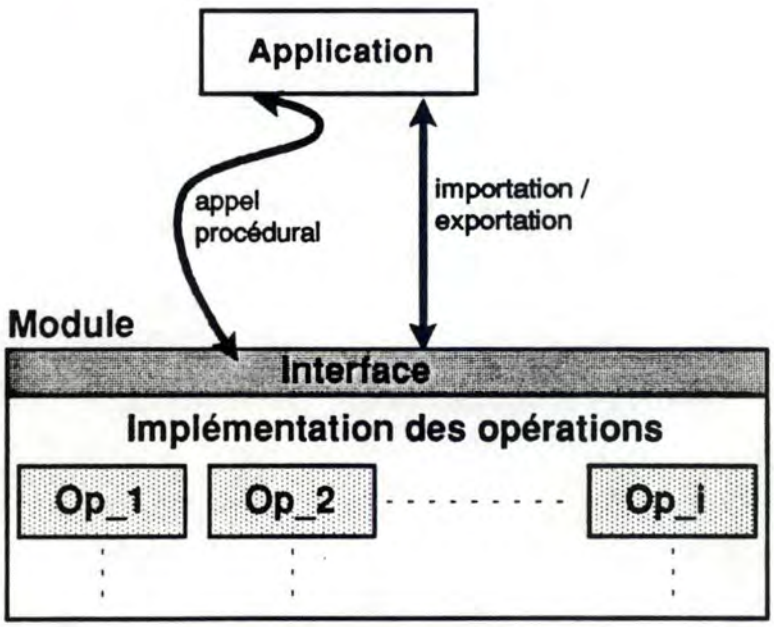

Schéma 3.3 : Le concept de module

37

3.3 Différences entre les appels locaux et les RPC

Les RPC entraînent l'échange de messages entre le client et le serveur. Ces messages doivent transiter par un réseau non fiable avec des ordinateurs et des logiciels qui sont tous susceptibles de tomber en panne ou de ne pas remplir convenablement leur fonction. Les machines et les logiciel qui tournent dessus ont des durées de vie variables et existent dans des espaces d'adressage isolés.

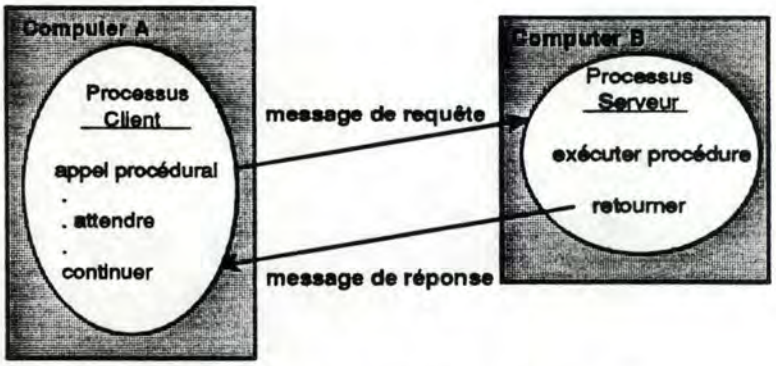

Schéma 3.4 : Le Remote Procedure Call

Ceci pose des problèmes pour le passage de paramètres contenant des adresses (*passage par référence*). Il est très difficile d'intégrer le concept de mémoire partagée entre machines incompatibles et lointaines dans un modèle de RPC et peu d'implémentations en tiennent compte. Le programmeur est souvent tenu à éviter les arguments en entrée/sortie contenant des adresses.

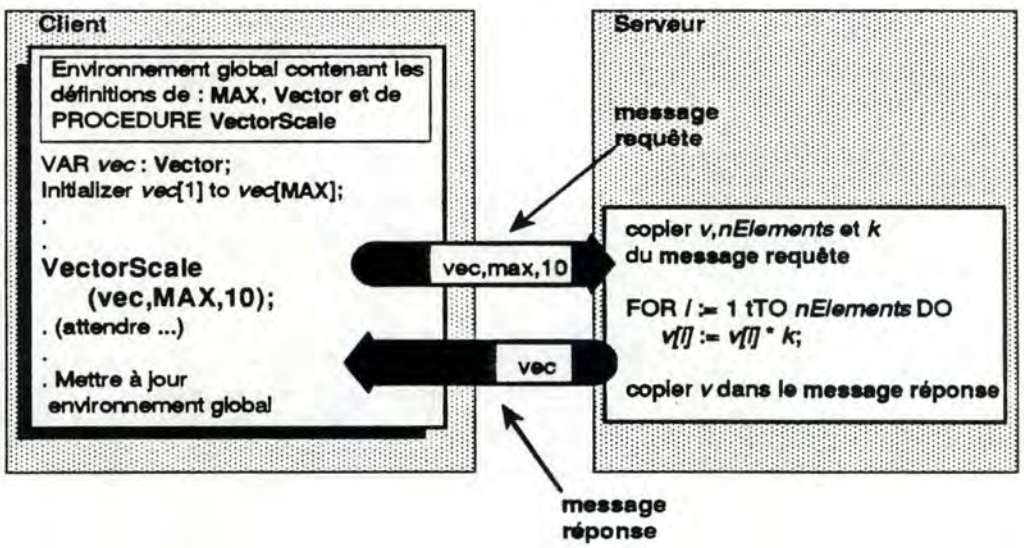

Schéma 3.5 : Diagramme des environnements pour un appel procédural lointain à VectorScale

Autres problèmes : Comment détecter des pannes et comment réagir à leur survenance? Avant d'effecteur le RPC le client doit localiser le serveur qui lui convient. Ce mécanisme est connu sous le nom de *binding* et pose de multiples problèmes : Comment localiser un serveur, que faire si le serveur n'existe plus, n'est pas en état de fonctionner ou a tout simplement migré sur une autre machine ?

38

Derrière une façade similaire aux appels locaux les RPC nécessitent des traitements bien plus complexes et sont sujets à la survenance d'événements qui ne concernent point les procédures locales. Ces traitements doivent être cachés aux programmeurs et aux utilisateurs pour faciliter leur tâches respectives.

4. Le paradigme RPC

4.1 Schéma descriptif

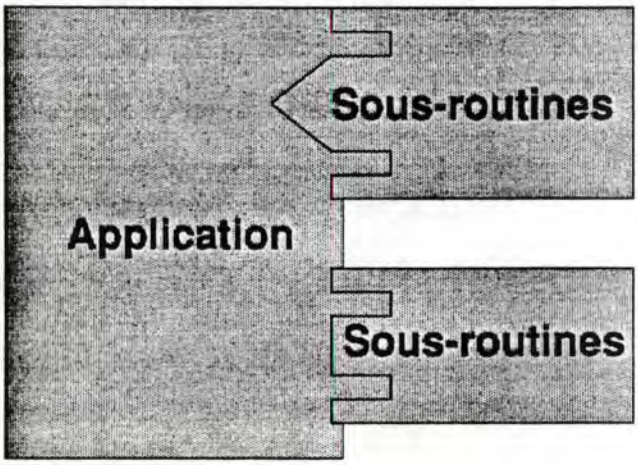

Schéma 4.1 : Appel procédural local

Ce schéma illustre l'appel de procédure quand l'application et les procédures se trouvent sur la même machine. L'application effectue un *call* à la procédure, celle-ci fait son travail et effectue un *return* vers le client. L'interface entre le client et la procédure est fixe et bien définie.

Dans le cas où les procédures et les clients se trouvent sur des machines différentes la situation change. Pour garder une certaine transparence, sans devoir changer le code des applications ou celui des procédures, le client exécutera les procédures effectives en passant par les procédures *stubs*. L'interface entre l'application et le serveur ainsi que le flux apparent des données resteront identiques.

4.2 Explications

Le schéma 4.3 nous montre le flux réel et le flux apparent des données lors d'un RPC. Nous y retrouvons le *client*, l'application appelante, et les *manager procedures* qui ne sont d'autres que les procédures locales du schéma de l'appel procédural classique. La séparation des clients et des procédures lointaines nécessite la création de *processus client* et de *processus serveur*, qui cachent l'existence des entités telles que les *stubs* et le module *RPC runtime*. Les paragraphes suivants expliqueront la fonction de ces entités logicielles et leur interaction lors d'un *Remote Procedure Call*.

39

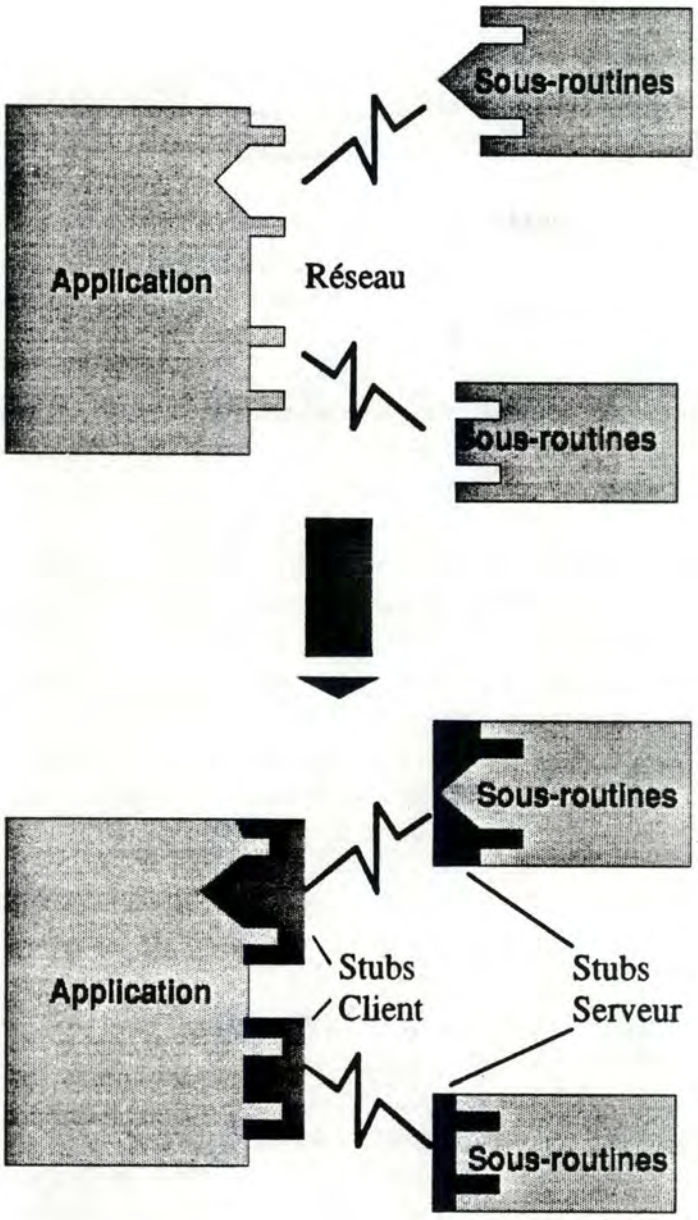

Schéma 4.2 : Extension de l'appel procédural local au RPC

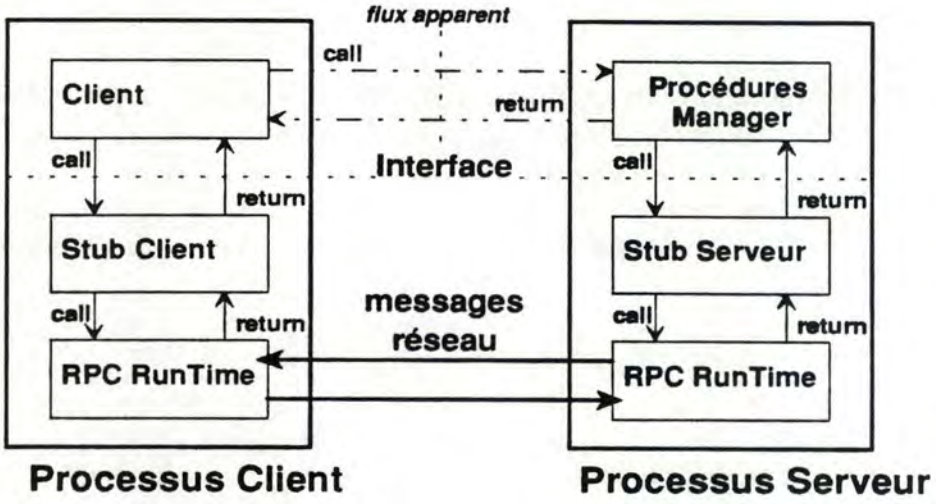

Schéma 4.3 : Le paradigme RPC

40

4.2.1 Les Clients et les Serveurs

Nous avons vu que l'application et les procédures du schéma d'origine ont été gonflées en *processus client* et *processus serveur*.

Le processus client s'occupe de transmettre l'appel de l'application à travers un réseau vers une procédure qui se situe sur une machine distante. Cette machine peut évidemment être incompatible avec la machine d'origine, avoir une autre architecture ainsi qu'une autre représentation des données.

Le processus serveur encapsule les procédures faisant partie d'un module. Il attend l'appel d'un processus client pour faire exécuter la procédure manager concernée et pour retransmettre au client appelant les résultats éventuels de cette exécution.

Les clients et les serveurs communiquent par messages qui passent par le réseau. Selon le mode de communication, fiable ou non-fiable, des messages comprennent plus que les informations concernant la procédure et les paramètres de celle-ci. Si l'interaction client-serveur se fait au moyen de datagrammes, des messages de gestion et de contrôle sont inévitables pour savoir ce qui est arrivé aux paquets qui transitent sur réseau. Voici quelques aspects auxquels le protocole RPC devra répondre : Le serveur a-t-il reçu l'appel ? Dans quel état (libre, occupé par mon appel, en panne ...) se trouve-t-il ? Le client a-t-il reçu les résultats? Est-il autorisé à recevoir les informations demandées ?

4.2.2 Les stubs

Les stubs sont les garants de la transparence des RPC. Ils existent du côté client (*client stub*) et du côté serveur (*server stub*).

Le ***client stub*** est une procédure locale qui possède une interface identique à la procédure lointaine concernée. L'application appelle le stub en pensant qu'il s'agit de la "vraie" procédure. La fonction du *client stub* est de capter les appels de l'application distribuée, de mettre en forme les arguments (*marshalling*) pour ensuite passer le tout au module *RPC Runtime*. Il attendra ensuite l'arrivée d'un message de réponse du RPC Runtime qui lui fournit ou bien les résultats de la procédure lointaine exécutée ou bien un message d'erreur. Des résultats seront ensuite mis dans une format local et retournés à l'application qui reprend son exécution.

Le ***server stub*** est appelé par le *RPC RunTime* lors de l'arrivée d'un message de requête RPC en destination du processus serveur. Il va prendre en charge ce message, en ressortir l'identification d'une fonction précise ainsi que ses arguments respectifs. Après avoir mis les arguments en une forme acceptable pour la machine locale (*marshalling*) la procédure concernée sera exécutée et ses résultats ou messages d'erreur retransmis au RPC RunTime.

4.2.3 Le RPC Runtime

La tâche du module RPC Runtime est la gestion de l'échange des messages entre processus client et serveur.

41

Schéma 4.4 : Le module RPC RunTime

C'est ici que va se dérouler le protocole RPC responsable

. du **traitement des messages du réseau** qui lui sont destinés. Ces messages pouvant être des requêtes ou des réponses, résultats ou erreurs, aux requêtes.

. de la **transmission de messages sur réseau** vers les autres acteurs lors d'un RPC. C'est le stub client (requête) ou serveur (réponse) qui demande cette transmission.

. du **traitement suite à des exceptions**, erreurs de transmission ou pannes de hardware ou de software.

4.2.4 Déroulement d'un RPC

Le déroulement d'un RPC est le suivant :

1. L'application client fait appel au stub client pour lancer un appel de procédure lointain

2. Le stub client met les données à transmettre dans une forme acceptable au RPC RunTime c'est-à-dire il linéarise les structures complexes (marshalling) avant de les passer au RPC RunTime.

3. Le RPC RunTime du client utilise les couches inférieures du logiciel de communication pour transmettre le message de requête à travers le réseau vers la machine concernée.

4. Le logiciel de communication fournit au RPC RunTime les messages en provenance du processus client. Après vérification de l'intégrité de ces données ceux-ci sont fournis au server stub.

5. Le server stub désassemble (unmarshalling) le message arrivé en ses différentes composantes comme l'identification de la procédure à exécuter et ses paramètres en entrée. Ensuite il fera un appel procédural local à la procédure manager concernée.

6. Le serveur exécute le corps de la procédure lointaine. L'exécution achevée la procédure passe le contrôle ainsi que les paramètres en sortie au server stub.

7. Le server stub linéarise la réponse de la procédure et donne celle-ci au RPC RunTime.

42

8. Le RPC RunTime transmet le message de réponse au processus client.

9. Le RPC Runtime du client captera le message de réponse pour le fournir au client stub.

10. Le client stub désassemble la réponse du processus serveur et enrichit l'environnement de l'application des résultats éventuellement fournis par la procédure lointaine.

11. L'application appellante reprend son exécution.

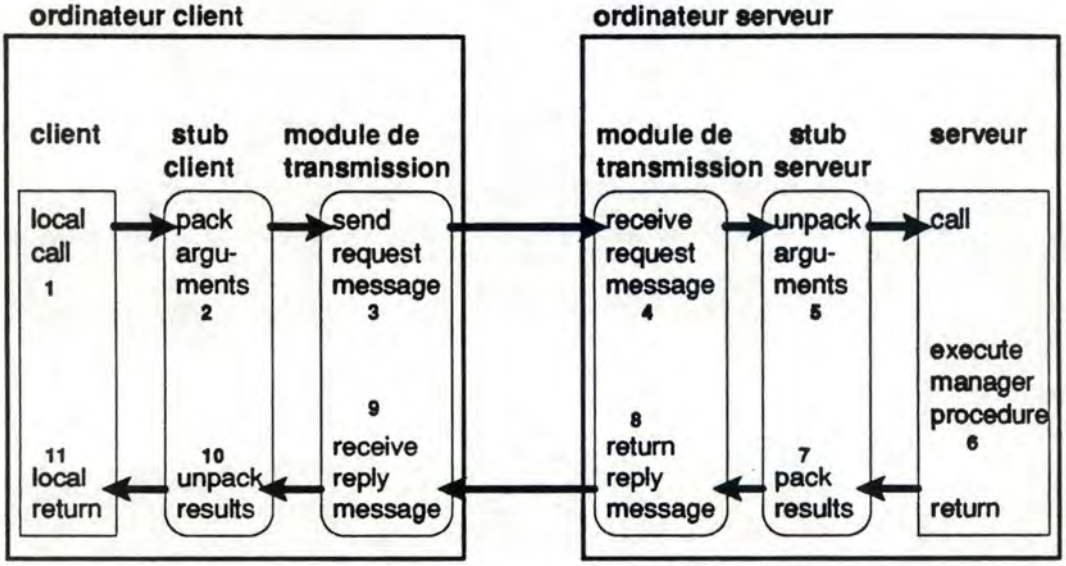

Schéma 4.5 : Déroulement d'un RPC

Cette description sommaire a cependant négligée quelques questions importantes :

- **Comment le client localise-t-il le serveur** qui encapsule le module et par conséquent la fonction lointaine à appeler ? Ceci constitue le problème du *binding* .

- **Quid de l'hétérogénéité des systèmes distribués ?** Pour permettre aux machines d'architectures différentes de communiquer il faut introduire des mécanismes de conversion de données. Ceux-ci feront l'objet du paragraphe 4.4 .

- **Comment garantir la transparence des appels lointains** par rapport aux appels locaux ? Le point 4.5 va élucider quelques aspects liés à cette transparence.

- **Quelle sera la sémantique des appels RPC ?** Trois types de sémantique sont présentés en 4.6 .

4.3 Le mécanisme de binding

Avant d'effectuer l'appel le client doit localiser le serveur qui exporte le module contenant la fonction appropriée. Ceci s'appelle l'*édition des liens* ou *binding*.

43

4.3.1 Le nommage d'un serveur

Un serveur encapsule les procédures d'un module. Pour ce module il va exporter un type de service avec une interface d'une certaine version. Chaque serveur possède un nom. Ceci peut être constitué par un numéro de serveur ou par le compostage du nom de la machine hôte, du type service, de l'interface ou de la version d'interface.
Voici quelques exemples de serveurs :

Type de service :	1. impression 2. accès base de données
Interfaces :	1. Postscript, ASCII, HP-GL 2. SQL, QBE, DBASE
Versions :	1. (Postscript) v44.56 (HPGL) 4.0 2. (SQL) 1.1 , (QBE) 1977 , (DBASE) III+

Noms :	"PRINT.PS.4456"	- impression postscript v44.56
	"DBA.DBASE.III+"	- accès à un BD sous DBASE III+
	"machineX:print/hpgl4.0"	- impression HP-GL sur machineX

4.3.2 La localisation du serveur

Plusieurs types de binding sont possibles : le binding *statique*, *dynamique* et le binding *sous contrôle de l'application.*

L'édition de liens statique n'est guère pratique puisqu'elle entraîne la recompilation de l'application distribuée suite à un mouvement ou tout autre changement des serveurs dont elle a besoin.

Le binding dynamique évite cette recompilation. Avant chaque appel remote le client tente de localiser le serveur qui dispose de l'interface et de la version approprié du module concerné.

Pour cela il s'adresse au ***binder*** ou ***éditeur de liens***. Le binder dispose d'une base de données contenant l'emplacement des différents serveurs, des types services qu'ils fournissent et des interfaces et versions disponibles. Ces informations lui sont fournis par l'opération d'*exportation* d'interfaces, réalisée par les serveurs. Pour savoir si un type de service avec une interface d'une version spécifique est disponible le client va consulter le binder.

Puisque le service de binding doit être fiable et disponible à tout instant il est conseillé de le répliquer à plusieurs endroits du système. Pour des raisons de sécurité le binder devrait être capable de cacher l'existence de certains service et d'interdire l'exportation de certaines interfaces. Il serait, en effet, trop facile d'exporter des interfaces d'authentification afin de piquer les mots de passe et les privilèges des autres utilisateurs.

44

Le binding dynamique, tel que décrit ci-dessus, peut devenir très coûteux et poser une charge lourde pour le réseau. C'est pourquoi on a envisagé une troisième solution : *le binding sous contrôle du programme*. L'idée est la suivante : le client localise le serveur avant le premier RPC et utilise le lien crée jusqu'au moment où des problèmes se présentent. A ce moment là l'application va réagir en analysant l'erreur produite et en prenant des mesures appropriées, telles que l'attente de la remise en marche du serveur ou le binding avec un autre serveur du même type.

4.3.3 Concordance des paramètres

Les types des arguments et des résultats donnés par le client doivent concorder avec ceux que spécifie l'interface de la procédure lointaine. Ce problème est résolu en compilant le client et le serveur à partir de la même définition d'interface. Les numéros de version des interfaces servent à garantir cette concordance.

4.3.4 Déroulement du binding

Le binding peut s'effectuer de plusieurs façons.

Le client peut connaître l'emplacement exacte du serveur et s'y adresser directement. L'adresse transport est donc connue en public (***well known port***). Cette adresse sera utilisée directement par le RPC Runtime pour réaliser l'échange des messages entre le client et le serveur.

L'autre façon, plus élégante, est de consulter le *binder*. Celui-ci se chargera de localiser le serveur et de fournir son adresse au client. Si plusieurs serveurs satisfont à la demande il peut en choisir un au hasard, ou bien selon des critères spécifiques (distance du client, charge des serveurs disponibles ...) ou bien tout simplement laisser le choix au client en lui offrant la liste des serveurs en question.

Lorsqu'un serveur est en panne, n'exporte plus la version X de l'interface I ou lorsqu'il est migré sur une autre machine le binding est devenu incohérent. Avant la mise à jour des tables du binder les clients ne sauront tourner convenablement. Réduire ce temps d'inconsistance à un minimum est l'objectif de tout binder qui se respecte. Le serveur ayant signalisé son nouveau profil et retiré toutes les informations obsolètes de la base de données du binder, le client pourra refaire le binding et continuer son travail.

4.4 Hétérogénéité des systèmes distribués

Les systèmes distribués peuvent être constitués de machine d'architectures tout à fait divergentes. La définition statique d'interfaces remotes et l'existence d'un format de données abstrait pour tous les ordinateurs du réseau sont indispensables pour leur interopérabilité.

45

4.4.1 Définition d'interfaces statiques

Une interface remote définit les caractéristiques des procédures fournis par un serveur et visibles à ses clients. Elle comprend les noms des procédures et la définition exacte de leurs paramètres en entrée et en sortie. Cette définition doit correspondre à celle utilisé par les clients.

Les clients et serveurs d'une application distribuée ont une existence propre. Il est dès lors impensable de faire des contrôles de consistance des accès aux interfaces lors de chaque appel de procédure, comme cela se fait dans plusieurs langages de programmation classiques. C'est pourquoi on définit les interfaces séparément dans un formalisme spécifique. Ceci est comparable aux *modules de définition* de MODULA2 où à la partie *Interface* des *units* qu'on trouve dans les versions récentes du TURBO PASCAL.

Le langage de définition des interfaces varie d'une implémentation à l'autre. Il fournit un certain nombre de types scalaires, types de base et des moyens pour en construire des types complexes, types structurés. Il doit permettre de définir clairement le type et la nature des différents paramètres : en entrée, en sortie ou les deux. Il est intéressant de disposer d'un langage basé sur les langages de programmation procéduraux. Certains systèmes fournissent plusieurs syntaxes de définition proches de langages de programmation connus, comme le C ou PASCAL.

Une fois la déclaration terminée un *compilateur d' interfaces* crée les procédures stubs pour le client et le serveur. Ayant connaissance de types de paramètres le compilateur prévoit le *marshalling*, la mise en forme de ceux-ci pour le service transport. En outre il s'occupe d'attribuer à chaque procédure faisant partie la définition un numéro d'identification ce qui permet au serveur d'effectuer le *dispatching*, la sélection de la procédure à exécuter.

4.4.2 La conversion des données

Dans un environnement hétérogène il est nécessaire de prévoir des mécanismes de conversion des données. Les types scalaires fournis par le compilateur d'interfaces sont représentés différemment sur les machines du réseau. Une vraie interopérabilité n'est possible que si cette situation est contrôlée. Plusieurs approches sont possibles :

1. **L'approche *canonique*.** Il existe un ensemble unique de types de données abstraits avec une représentation physique externe à toutes celles des machines physiques, connue sous le nom de *external data representation* (EDR). Lors de l'envoi des arguments ceux-ci sont convertis du format local en EDR pour ensuite être retraduits du EDR vers la représentation locale. Lorsque la communication se déroule entre machines du même type il devrait être possible d'éviter ces traductions inutiles et de se passer de l'EDR.

2. **Une approche *multicanonique*.** On fournit une liste de formats standardisés par plusieurs constructeurs ou organismes internationaux. Lors de l'envoi d'un message l'origine indique le format des données utilisé dans la liste des format disponibles. Il n'y aura donc plus de conversions à ce niveau. La traduction sera du domaine du destinataire du message. Si le format choisi précédemment ne lui convient pas il le traduira dans sa représentation locale.

46

Nous remarquons que l'approche multicanonique (***"receiver makes it right"***) est souvent bien plus efficace puisqu'elle évite des traductions inutiles mais cela se fait au prix d'un gonflement du software RPC qui devra dès lors prévoir une multitude de représentations diverses. Cette approche a pourtant le désavantage d'isoler certaines architectures peu communes, qui n'utilisent aucune des représentations proposées.

4.5 La transparence des RPC

Rendre l'utilisation du RPC confortable pour attirer les programmeurs d'applications implique la dissimulation des mécanismes complexes qui se cachent derrière une syntaxe identique à l'appel procédural classique. Or nous avons vu que l'appel de procédures lointaines se heurte à des problèmes supplémentaires résultant de l'utilisation de canaux de communication non-fiables, de la diversité des architectures des différents ordinateurs et surtout du fait que les processus clients et serveurs ont des espaces de mémoire non partagés et possèdent des durées de vie différentes.

4.5.1 La sémantique du RPC

Un appel lointain n'est considéré comme terminé que du moment que le client l'ayant initié a reçu du serveur la réponse respective. Or entre ces deux situations plusieurs pannes peuvent survenir :

1. **La perte du message de requête**
2. **La perte du message de réponse**
3. **Le serveur tombe en panne et redémarre**
4. **Le client tombe en panne et redémarre**

La sémantique du RPC est déterminé par la réaction du software RPC à la survenance de ces pannes.

La sémantique ***"maybe"*** est caractérisée par le fait que l'application appellante ignore les pannes et que la non-exécution de la procédure appelée n'a pas de conséquences sur son déroulement. Le client lance l'appel RPC et continue son travail. Dans ce cas-ci l'appel ne devra pas contenir des paramètres en sortie, puisque la valeur de ceux-ci est arbitraire.

La sémantique ***"at least once"*** prévoit l'exécution de la procédure appelée et garantit la réception des résultats éventuels par le client. Elle entraîne l'attente du client jusqu'à la réception des résultats. Pour éviter une attente infinie le software RPC du client envoie, après un délai de *timeout*, des message d'enquête quant à l'état de l'appel en cours. Suite à cela le client peut détecter des pannes de réseau ou des ordinateur communicants et réagir convenablement.

Or cette dernière sémantique ne considère pas le nombre de fois que la procédure sera exécutée. On peut imaginer la surprise d'un client d'une banque lorsque son compte est débité trois fois de 10.000 BEF suite au retrait unique de cette somme.

47

La sémantique "***at most once***" reprend les fonctionalités du "at least once" et en plus elle garantit la non-exécution ou l'exécution unique de la procédure appelée. C'est la sémantique la plus riche du RPC et en même temps la plus lourde à implémenter.

4.5.2 Le traitement des exceptions

Le serveur doit avoir la possibilité de signaliser des erreurs ou incohérences survenus au client appelant. Le software RPC doit pouvoir signaler à l'application concernée, selon le protocole choisi, les pannes de réseau ou des ordinateurs communicants ainsi que le non respect de la syntaxe choisie. Pour signaliser ces erreurs il va utiliser le mécanisme de l'indication des exceptions (***exception raising***). Le client s'occupera de capter l'indication de l'exception et de lancer une procédure locale de gestion d'erreur (***exception handler***). L'indication d'exception se fait le plus souvent par la modification d'une variable de l'environnement du client. Le client consultera la valeur de la variable et fera dans le code du client un appel vers le *handler* approprié. Une autre possibilité est l'exécution automatique suite au retour d'un *signal d'exception*. Ceci ressemble plus au mécanisme de signalisation inter-processus connus dans les systèmes multi-programmes classiques mais requiert l'existence de mécanismes de notification et de gestion d'exceptions dans les langages de programmation utilisés dans le développement de l'application.

4.5.3 Le passage des paramètres

Les espaces de mémoire étant disjoints pour le client et le serveur le passage de paramètres par référence devient très difficile. Une adresse qui pointe vers une zone significative pour le client peut pointer chez le serveur vers une zone protégée ou une zone contenant des valeurs arbitraires. Normalement la procédure *stub* se charge de rechercher la contenu de la zone pointée et l'utiliser en paramètre d'entrée ou de sortie. Or si cette zone est complexe et comporte des nouveaux pointeurs la situation se complique. Comment utiliser des listes chaînées ou d'autres type structurés contenant des pointeurs, dans un RPC? Que faire des pointeurs génériques ?

Garantir une transparence suffisante pour ces cas n'est pas une tâche facile et complique la mise au point du compilateur d'interfaces. Une solution radicale serait d'interdire aux programmeurs l'utilisation de pointeurs. Mais la transformation d'applications existantes en applications distribuées risque alors de coûter bien trop cher. Le compilateur peut alors prévoir la possibilité d'écrire des procédures de transformation des structures complexes garnis de pointeurs de toute sorte vers des structures gérables par le système RPC disponible. Chaque référence à ce type "*compliqué*" entraînera le lancement d'une fonction de "*simplification*" dont le résultat est transmissible par les stubs créés.

48

5. Le software RPC

Par software RPC on comprend les éléments suivants :

1. **Le module *RPC RunTime*,** qui s'intègre dans le modèle hiérarchique du software de communication. Il propose une série de services accessibles aux clients et aux serveurs et fait usage des fonctionalités des protocoles de transport existants pour établir une communication entre eux.

2. **Les *stubs* du client et du serveur,** crées par le compilateur d'interfaces et linkés avec l'application et les serveurs de l'application distribuée

3. **Le *compilateur d'interfaces*,** qui prend en entrée la définition des interfaces lointaines et crée en contrepartie les procédures stub qui seront intégrés au schéma de compilation lors de la compilation des applications.

Ses tâches principales sont dès lors de :

1. **transmettre des messages RPC** de requête et de réponse à travers le réseau en utilisant un protocole de transmission de messages RPC

2. **linéariser et délinéariser les arguments** à transmettre et les réponses reçus et de **sélecter dans le serveur la procédure à exécuter**

3. faciliter la création d'applications distribués par l'**intégration aisée des modules RPC dans les langages de programmation répandus**

5.1 La transmission de messages par réseau

Le mécanisme RPC se trouve souvent basé sur la couche de transport du logiciel de communication. Il reprend et simplifie le travail fait par les couches supérieures du modèle OSI [Tanenbaum]. Un appel lointain est accompagné par l'échange de messages entre un client et un serveur. Cet échange se fait selon un protocole RPC bien défini. Il fait usage des services de transport fournis tels que les opérations *SendMessage* et *ReceiveMessage* selon un protocole de transport. Les points suivants vont élucider le choix de ce protocole de transport, la composition des messages échangés ainsi que les différents protocoles RPC existants.

5.1.1 Le choix du protocole de transport

La couche transport dispose de deux modes de communication. Il s'agit d'un mode orienté connexion , celui du circuit *virtuel* et d'un autre mode orienté non-connexion par *datagrammes*.

49

Un ***circuit virtuel*** consiste en une connexion permanente entre deux unités communicantes. Avant de communiquer suite à l'obtention d'un circuit virtuel il faut ouvrir la connexion et négocier un certain nombre de paramètres. La fin de la connexion est accompagnée d'une fermeture du circuit virtuel. Le circuit virtuel est comparable au mécanisme de *pipe* de UNIX [Unix]. Un flot de données passe en séquence à travers le circuit virtuel d'une origine à une destination. La consistance et l'ordre de transmission des données est garantie par ce mode de communication.

La ***communication par datagrammes*** ne garantit ni l'arrivée à destination des messages ni l'ordre de transmission. Il n'y a pas de connexion entre deux programmes donc pas d'ouverture ni de fermeture de celle-ci, ni de phase de négociation des capacités des unités communicantes. Chaque datagramme transmis comprend l'adresse de la destination et le système s'efforce de l'y faire parvenir sans garantie aucune.

Les protocoles RPC utilisent pour la plupart des cas le système des datagrammes. Les raisons sont les suivantes :

1. Les messages RPC sont en principe assez courts et l'établissement et la destruction de connexion sont des surplus gênants

2. Les serveurs doivent parfois s'occuper d'un grand nombre de clients et le stockage des informations d'état pour chaque communication en cours peut coûter cher

3. Les réseaux locaux sont plutôt fiables. La survenance d'erreurs étant assez rare l'existence d'un protocole de connexion sophistiqué est redondante.

4. Les protocoles de datagrammes se retrouvent sur la majorité des systèmes connus. On est presque toujours sûr de pouvoir communiquer par datagrammes, ce qui n'est guère le cas pour les circuits virtuels.

5.1.2 La composition des messages RPC

Le module de transmission, le RPC RunTime, gère le passage de messages entre clients et serveurs. La structure minimale des messages est la suivante :

```
TYPE message = RECORD
    messageType     : { Request, Reply };
    messageID       : INTEGER;
    requestId       : INTEGER;
    sourceAddress   : AddressType;
    procedureId     : INTEGER;
    arguments       : ByteStreamType;
    multiPacket     : BOOLEAN;
    seqNumber       : INTEGER;
END
```


50

messageType permet de distinguer entre les messages de requête et les messages de réponse.

Le champ ***messageId*** sert à reconnaître la réponse à un appel spécifique. Le client peut ainsi éviter des réponses erronées à des vieilles requêtes. Il sert aussi de support à la sémantique "at most once". Pour garantir qu'un appel n'est exécuté qu'une seule fois le serveur tient des tables des appels à effectuer et des appels en cours. Lorsqu'un client retransmet sa requête le serveur saura qu'elle a déjà été faite et ne répondra qu'à cette dernière requête. Il tiendra aussi une table des réponses aux dernières requêtes effectuées ce qui permet en cas de besoin de retransmettre une réponse éventuellement perdue.

Grâce au champ ***requestID***, généré par le client, le serveur peut filtrer les messages doubles.

L'adresse de l'origine du message (***sourceAddress***) est indispensable au serveur pour retourner les résultats éventuels d'une exécution. Selon le type de système distribué et de réseau cette adresse est représentée différemment. Elle contient l'identification d'un réseau, d'une machine spécifique et d'un *port*, point terminal de communication entre processus.

Le champ ***procedureId*** identifie la procédure dont le corps est à exécuter. Comme une interface peut comprendre plusieurs procédures on leur attribue un numéro d'identification qui servira lors du *dispatching*.

Les ***arguments*** de l'appel sont représentés en listes aplaties (*marshalled*) afin de pouvoir être transmis par un canal de communication série.

Voyons ici un exemple de paquet RPC :

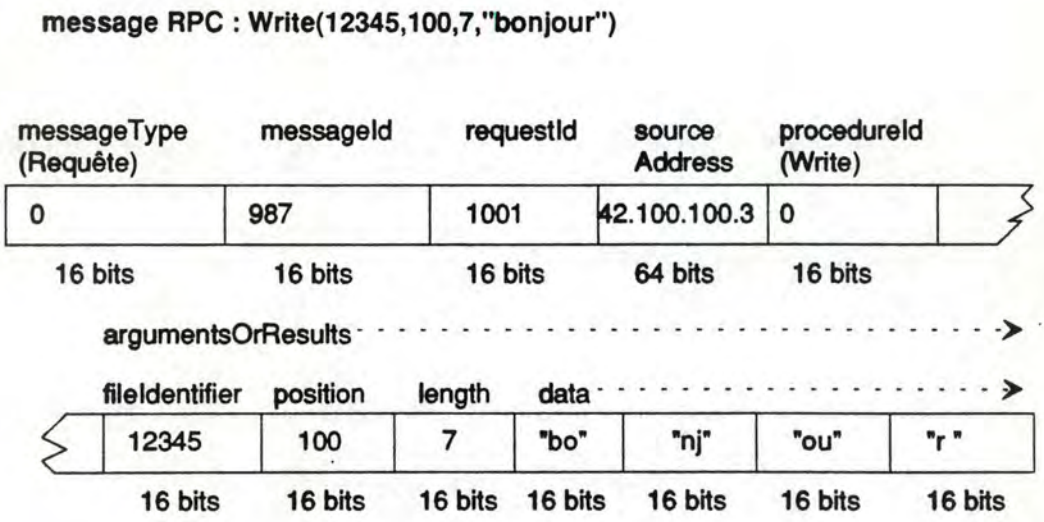

Schéma 5.1 : Message de requête RPC

51

5.1.3 Les protocoles RPC

Le schéma suivant reprend sous forme d'une table les trois protocoles disponibles ainsi que la source des divers messages envoyés :

Nom du protocole	Origine des messages		
	Client	Serveur	Client
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge-Reply*

Schéma 5.2 : Les protocoles RPC

Le protocole ***Request*** (R) suffit comme support de la sémantique "maybe". Ce protocole est le moins fiable puisque aucun contrôle d'erreur n'est fait et aucune confirmation n'est envoyée aux clients. Il trouve souvent son utilisation dans les serveurs graphiques de fenêtres [XWindows].

Le protocole ***Request/Reply*** (RR) est déjà plus fiable. Les clients, avant de poursuivre leur exécution, attendent la réponse, souvent accompagnée de résultats, du serveur. Les sémantiques "at least once" et "at most once" sont couverts par ce protocole.

La sémantique "*at most once*" nécessite la tenue de tables de requêtes et de réponses du côté du serveur. Le serveur garde les réponses dans une table et en cas de besoin il retransmet ces réponses au client. Or il est difficile de prédire quand l'appel lointain est terminé. Faut-il garder la réponse pendant 1 seconde où 10 secondes ? La garder trop longtemps serait du gaspillage ! C'est la raison pourquoi l'on trouve parfois des protocoles Request/Reply enrichis d'une confirmation de réception des réponses du client. Il s'agit du protocoles RRA, ***Request/Reply/Acknowledge-Reply***.

Les opérations transport *SendMessage* et *ReceiveMessage* servent à transférer un paquet d'un taille maximale (de l'ordre de quelques KB) d'une origine à une destination. Pour la transmission de messages excédant cette longueur une couche intermédiaire entre le transport le software RPC devient nécessaire. Cette couche s'occupe de trancher les messages longs en *multipaquets à l'envoi* et de les réassembler à la réception. Pour distinguer des messages simples et des multipaquets la structure des messages est enrichie des champs *multiPacket* et *seqNumber* qui indiquent s'il s'agit d'un multipaquet et si oui de son numéro de séquence.

Selon le type de conversion de données qu'on trouve on peut ajouter à la structure du message des identifiants du format des données utilisé.

52

5.2 Le marshalling et le dispatching

Le ***marshalling*** consiste dans l'assemblage des données en une forme qui convient à la transmission à travers le réseau lors de l'envoi et dans le désassemblage de ces mêmes données lors de la réception.

Avant de transmettre les données ils doivent être linéarisés afin d'être intégrés dans des messages sous forme de listes aplaties (*bytestreams*). Voyons un exemple :

Marshalling des arguments de : VectorScale(vec,MAX,10)

```
CONST MAX=5;
TYPE  Vector = ARRAY[1..MAX] of INTEGER;
VAR   vec : Vector;
```

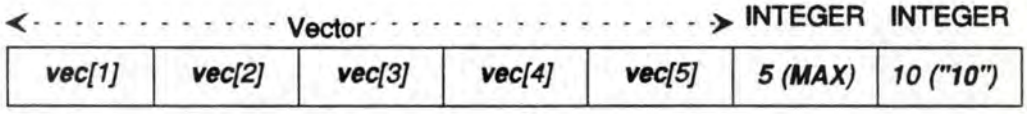

Schéma 5.3 : Exemple de marshalling (linéarisation)

Le software RPC fournit des procédures de marshalling pour la plupart des types simples connus ainsi que des procédures de transformation de types composés vers ces types scalaires. Dans certaines implémentations il est laissé au programmeur de *marshaller* les arguments. Ceci est utile si les types sont trop complexes ou comportent des types scalaires inconnus ou interdits (pointeurs).

Le *dispatching* consiste dans la sélection de la procédure à exécuter et du lancement de celle-ci. Pour savoir quelle procédure de l'interface est à exécuter le *stub* n'aura qu'à consulter le champ *procedureId* du message de requête reçu. Ensuite il lancera la procédure appropriée.

Le processus qui exécute l'appel remote a une durée de vie indépendante de celle de l'application appellante. Le déroulement de l'exécution peut se faire de deux façons.

1. Pour chaque client ou pour chaque requête un processus d'exécution RPC est lancé. Ce processus est tué par le système à la fin de son exécution.

2. Un processus, gérant des appels RPC, tourne de façon continue. La plupart du temps il se trouve dans une boucle d'écoute. Si une requête lui parvient il exécute dans son espace de mémoire la procédure choisie et retourne ses résultats à la fin. Ceci permet le partage de certaines zones de mémoire du serveur par tous les appellants.

53

6. Les aspects de performance des RPC

Pour trouver une utilisation répandue une implémentation du software de support du RPC doit être performante. Les programmeurs ne vont pas utiliser le RPC pour la beauté du geste et une implémentation lente et inefficace ne fera que décourager son utilisation.

6.1 Les mesures de performance

Avant de penser à une optimisation quelconque il faut analyser les différents facteurs qui ont une influence sur le temps pris par un RPC.

1. le marshalling des arguments et des résultats.
2. la transmission à travers le réseau (bidirectionnelle)
3. le temps d'exécution de la procédure dans le serveur
4. le temps d'attente avant l'obtention du service demandé.

Augmenter les performances des RPC équivaut à optimiser les quatre facteurs listés ci-dessus.

1. Optimiser le marshalling équivaut à optimiser les procédures de marshalling de façon à ce qu'elles exécutent convenablement et rapidement leur travail.

2. Le temps de transmission est composé du temps passé dans le software de communication et du temps utilisé par le support physique pour moduler et transmettre le message d'une machine à l'autre. Ce dernier temps dépend du type de réseau, de sa largeur de bande ainsi que du nombre d'utilisateurs qui se partagent les canaux de communication.

L'optimisation d'un système existant joue donc surtout au niveau du logiciel de communication. Les objectifs primordiaux sont la réduction du nombre de modules et de la quantité de données échangés entre eux, une simplification des protocoles intermédiaires et une diminution de la taille des paquets transmis au niveau physique. Le choix du protocole de transport n'est pas à négliger non plus. Il serait intéressant de laisser ce choix au programmeur. Si les arguments d'un appel sont de taille considérable il peut redevenir intéressant d'utiliser les circuits virtuels au lieu des datagrammes.

3. L'exécution rapide de la procédure lointaine évite le blocage inutile du serveur et le rend vite disponible au client suivant.

4. Il s'agit ici du temps passé dans des files d'attente du serveur ainsi que du temps nécessaire au système pour la création de l'espace d'exécution et le lancement du processus d'exécution du corps de la procédure lointaine.
Plusieurs serveurs du même type augmentent la disponibilité d'un service et évitent des files d'attentes excessives. Si chaque appel RPC résulte dans la création d'un nouveau processus il faut prendre en compte le temps de lancement de celui-ci et la charge posée sur le système qui devra gérer tous ces processus. Si une machine est l'hôte de plusieurs serveurs très demandés la gestion des processus, dont chacun possède son propre espace d'adressage virtuel ainsi que les changements de contexte risquent de créer un overhead considérable pour tout appel de procédure lointaine.

54

L'existence de serveurs qui tournent parallèlement et en permanence évite déjà la création de nouveaux processus à chaque RPC. Pour faciliter la communication entre ces serveurs le système devrait disposer de puissants mécanismes de communication ainsi que de processus légers, avec une temps de création et de mise en marche minimal.

6.2 Implémentation du RPC par les processus légers (*threads*)

6.2.1 Prérequis théoriques

Dans les systèmes d'exploitation basés sur UNIX les processus sont tout à fait indépendants. Ils possèdent chacun son propre espace d'adressage et communiquent par *pipes* ou par le mécanisme de *sockets* [Unix]. Chaque communication inter-processus entraîne un double transfert de zones de mémoire. La zone d'origine est copiée dans un tampon système pour être ensuite recopiée dans l'environnement du processus destinataire. Si ce modèle est encore appliquable pour les ancien systèmes à temps partagé (*timesharing systems*) il ne semble guère adapté aux systèmes distribués. Il ne s'agit évidemment pas de laisser tomber toute protection des processus mais plutôt de permettre le partage de mémoire entre certains processus auxquels on peut faire confiance.

Un serveur peut alors être imaginé comme processus qui tourne en permanence et qui, lors d'une RPC lance le ***thread*** qui correspond au processus d'exécution de la procédure appelée. Pour cela nous utiliserons l'idée de ***moniteur*** [Hoare].

Un moniteur est un objet logiciel qui réunit plusieurs variables partagées, un mécanisme d'exclusion mutuelle pour protéger l'accès à celles-ci ainsi que des procédures qui les utilisent lors de leur exécution. Le moniteur comprend donc :

- des déclarations de variables à partager
- la définition de **procedures d'entrée** (*entry procedures*) qui sont appelés de l'extérieur et ont des incidences sur les variables partagées
- la déclaration de variables non partagées et de procédures locales

Pour permettre la synchronisation des accès aux zones communes les règles suivantes sont imposés par le moniteur :

1. Seul un processus à la fois peut entrer dans le moniteur, c'est à dire exécuter l'une de ses procédures. Les autres processus seront mis en attente jusqu'à ce que le moniteur soit de nouveau disponible.

2. Tout processus peut appeler les procédures d'entrée

3. Les variables partagées par le moniteur sont uniquement accessibles aux procédures du moniteur. Il est recommandé que ces procédures ne font pas usage de variables externes au moniteur.

55

Un processus *entre* dans le moniteur suite à l'appel d'une procédure d'entrée. Il en *sort* lors du retour de celle-ci. Entre ces deux moment il se trouve *dans* le moniteur. Il peut être en train d'exécuter une procédure ou bien être suspendu et se trouver dans une file d'attente. Du moment qu'une procédure d'entrée se termine le moniteur prend l'entrée suivante, si elle existe, et exécute la procédure demandée par celui-ci. L'exclusion mutuelle des processus dans le moniteur est réglée par le mécanisme des ***sémaphores*** [Unix].

Les moniteurs disposent d'une possibilité de signaler des événements d'un processus à un autre. Les ***variables partagées conditionnelles*** permettent à un processus de suspendre son exécution et de redonner la main aux autres. Si une certaine condition est vérifiée, si un processus met à jour la variable conditionnelle, le processus interrompu reprend son travail. Les primitives *wait* et *signal* gèrent la synchronisation d'exécution de processus collaborants.

La primitive ***wait***(*s*), où *s* est une variable conditionnelle, interrompt le processus en cours d'exécution, le met à la fin de la file d'attente, et repasse la main au premier en attente dans la file.

La primitive ***signal***(*s*), *s* étant une variable conditionnelle, relance le premier processus en attente de la modification de la variable *s* et le sort de la file d'attente. Le processus ayant utilisé cette primitive est alors mis en attente jusqu'à la fin de cette exécution.

6.2.2 L'implémentation du RPC par moniteurs

a) *Le côté serveur.*

Le serveur dispose d'un processus d'écoute (***distributor***) qui capte les requêtes de clients éventuels, d'un ensemble de processus légers de travail (***workers***) qui exécutent les procédures appelées et d'une mémoire qu'ils se partagent tous.

Au moment du lancement du serveur les workers se trouvent en attente de travail suite à l'appel de la primitive *wait*(*C*). Le distributeur met les messages de requête dans une zone tampon partagée (***CallsIn list***) par tous les workers et signale leur arrivée par *signal*(*C*). Selon le mécanisme du moniteur que l'on vient de décrire un des workers libre reprendra son exécution. Il va prendre le premier message de la file de messages, l'analysera, exécutera la procédure demandée et renverra les résultats éventuels au client respectif. Par la suite il fera de nouveau un *wait*(*C*).

Le moniteur s'occupe de la gestion de la file des messages afin que les ajouts du distributeur et les retraits des travailleurs sont sérialisés. Reste encore à enlever l'enregistrement de la requête exécutée de la liste des appels en entrée.

b) *Le côté client.*

Du côté client l'on dispose d'un modèle semblable. Lorsque plusieurs processus clients sont en cours d'exécution il devient intéressant de les regrouper comme le montre le schéma ci-dessous.

56

On y trouve les processus clients (*workers*), un distributeur et une zone de mémoire commune. Les workers, lors du RPC, mettent une copie du message de requête dans le buffer des appels sortis (***CallsOut list***) et attendent les résultats en faisant un *wait(C)*. Le distributeur reçoit les réponses des serveurs, copie les résultats éventuels vers l'enregistrement correspondant dans la liste des messages en sortie et signale leur arrivée au worker concerné. Celui-ci reprend alors son exécution en transférant les résultats du RPC vers son espace d'adressage. Evidemment la liste des appels en sortie sera amputée de l'enregistrement correspondant au RPC considéré comme terminé.

c) Le déroulement du RPC

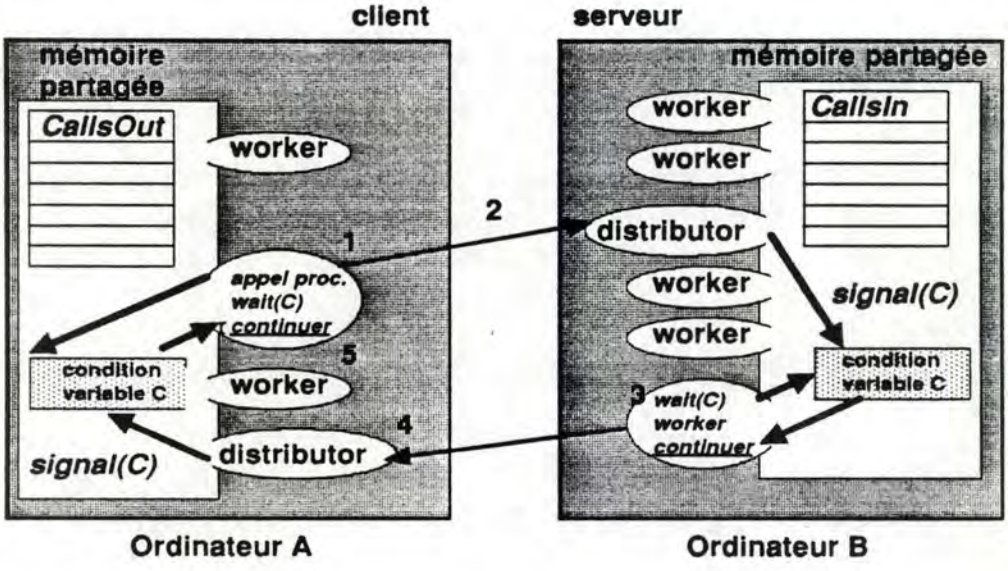

Schéma 6.1 : Implémentation du RPC par processus légers

La séquence d'événements,illustrée dans le schéma 6.1, lors du Remote Procedure Call est alors la suivante :

1. Le processus client dans l'ordinateur A fait un appel à la primitive *DoCall.*

resultats := *DoCall*(*destination, procId, dataIn*)

Les actions de *DoCall* sont :

(a) Créer un nouveau message et remplir ses champs

(b) envoyer le message à sa destination (Machine B)

(c) ajouter message envoyé à la liste *CallsOut*

(d) attendre le *signal* généré par l'arrivée d'une réponse

57

2. Le message envoyé en 1. est reçu par le distributeur dans la machine B qui :

(a) reconnaît qu'il s'agit d'une requête

(b) exécute *StartCall*(*msgReq*)

- en plaçant le message reçu (*msgReq*) dans la liste *CallsIn*
- en envoyant un signal qui relance l'exécution d'un *worker*

3. Le *worker* dans la machine B :

(a) appelle *getMessage* pour obtenir le message de la liste *CallsIn*

(b) exécute la procédure lointaine demandée et fabrique un message de réponse, *msgRep*.

(c) lance *EndCall*(*msgRep*) qui

- renvoie *msgRep* au client
- si la liste *CallsIn* est vide effectue un *wait*
  sinon retourne au point 3.

4. Le distributeur dans la machine A

(a) reçoit le message de réponse *msgRep*

(b) retrouve le message de requête correspondant dans la liste *CallsOut* et le remplace par le message de réponse obtenu

(c) signale au processus *worker*, origine de la requête, que la réponse et arrivée

5. Le processus worker sort la réponse de la liste *CallsIn* et reprend son exécution.

58

Chapitre III. NCS - Network Computing System

1. Introduction

Le NCS, *Network Computing System*, est un ensemble d'outils facilitant le développement et l'exécution d'applications distribuées[NCS][NCA]. Il s'agit d'une implémentation de l'architecture NCA (*Network Computing Architecture*) mise au point par Hewlett Packard et APOLLO. NCA est une architecture offrant des facilités de distribution d'applications sur une collection hétérogène d'ordinateurs, de réseaux et d'environnements de programmation. Les applications bâties sur NCA ont leurs données ainsi que leurs traitements répartis sur des machines divergentes, ceci dans le but d'optimiser l'utilisation des ressources du réseau en fournissant un service de qualité aux utilisateurs.

1.1 Les principes du NCS

- Ouverture, portabilité et hétérogénéité

La spécification du NCA est publique. Elle est accessible à toute personne intéressée. Le code source du NCS est disponible et cela gratuitement pour toute institution à but non lucratif ou université. NCS est rédigé entièrement dans le langage C. Le protocole RPC, le compilateur d'interfaces ainsi que les implémentations des clients et serveurs et des différents outils de gestion fournis avec le NCS disposent tous d'un code source en C. Puisque le langage C est disponible sur la majorité des systèmes informatiques et que l'on ait porté une attention particulière à la portabilité du code l'on trouve de plus en plus d'implémentations du NCA. On en trouve, par exemple, sous DOMAIN/IX, BSD-UNIX 4.2 et 4.3, UNIX System V.x, MS-DOS et même sous VMS.

Une caractéristique essentielle du NCS est son *orientation objet*. Tout élément du réseau, qu'il s'agisse d'un processus en exécution ou d'un canal de communication, est considéré comme objet.

NCA dispose d'un langage de définition d'interfaces, le **NIDL** (*Network Interface Definition Language*), d'un protocole **RPC** (*Remote Procedure Call*) et d'un format interne de représentation des données **NDR** (*Network Data Representation*) propre à lui. Les services intégrés dans l'architecture NCA sont accessibles à travers un ensemble d'interfaces définies en NIDL. Le *location broker*, serveur de noms pour les objets distribués, exporte une telle interface NIDL.

L'orientation objet, qui s'oppose à une *orientation serveur*, et l'existence d'un protocole de présentation de données facilitent l'implantation du NCA dans des réseaux composés de matériel hétérogène. NCA est indépendant du protocole réseau utilisé.

59

- *Transparence et performance*

Le protocole NCA/RPC est fondé sur le protocole de Birrell & Nelson [BiNe] décrit au chapitre II. L'objectif est de maximiser la ressemblance entre les appels locaux et les appels remotes de procédures. L'usage des RPC est transparent au programmeur et à l'utilisateur.

NCA/RPC dispose d'un protocole léger *request-response* qui utilise le service datagramme de la couche transport. Le protocole se sert de processus légers, des *threads*. [NCA_CPS]

La conversion des représentation des données ne se fera que si le besoin se présente. L'approche adoptée est *multicanonique*, c'est-à-dire que chaque paquet transmis possède un champ indiquant le format utilisé par l'origine. Si ce format diffère de celui du destinataire, celui-ci s'occupera de le transformer. Dans le pire des cas il y aura donc une et une seule conversion.

- *Extensibilité et sécurité*

NCA constitue le fondement pour le développement d'applications distribuées fiables et portables. Son architecture adresse tous les principaux problèmes rencontrés par les développeurs de telles applications. Il présente une architecture ouverte et aisément extensible. L'ajout de nouveaux outils ne pose aucun problème: Il suffit de mettre au point les serveurs désirés et de définir leurs interfaces en NIDL avant de les exporter aux éventuels clients.

NCA ne prévoit pas de mécanismes de protection dans son architecture de base. L'intégration d'outils de protection et de sécurité se fait à travers la mise au point de serveurs spéciaux. Ces *brokers* fournissent de services comme l'authentification ou la distribution de clés de chiffrage. Les détails concernant la standardisation de ces serveurs sont toujours sous étude.

1.2 L'architecture NCA

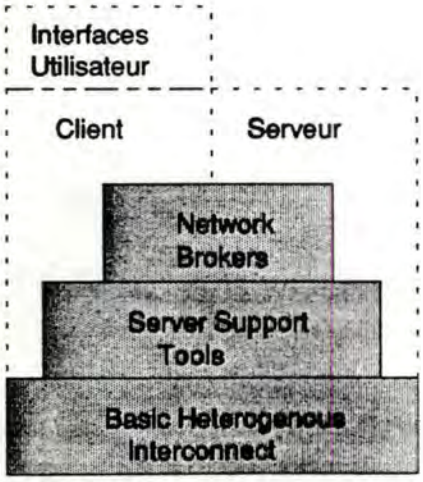

Schéma 1.1 : Architecture NCA

60

NCA applique le *modèle client-serveur* aux applications distribuées. Chacune de ces applications est constituée de programmes client, consommateurs de ressources, et de serveurs d'application, fournisseurs de ressources. Un serveur de fichiers s'occupe des opérations de manipulation de fichiers tandis qu'un autre serveur peut gérer des processeurs spécialisés dans les calculs vectoriels. L'accès aux programmes client se fait à travers les interfaces utilisateurs.

Les clients et les serveurs d'applications font un usage régulier des *network brokers*, des *server support tools* ainsi que du *basic heterogenous interconnect*.

1.2.1 Les Brokers

Un broker est un serveur qui fournit des informations au sujet des ressources du réseau, qu'elles soient locales à un host ou remotes. Il agit en tant qu'intermédiaire entre les clients, demandeurs de ressources, et les serveurs, fournisseurs de ressources. Le programmeur peut ainsi se concentrer sur les aspects d'exécution de l'application et non sur les aspects opérationnels.

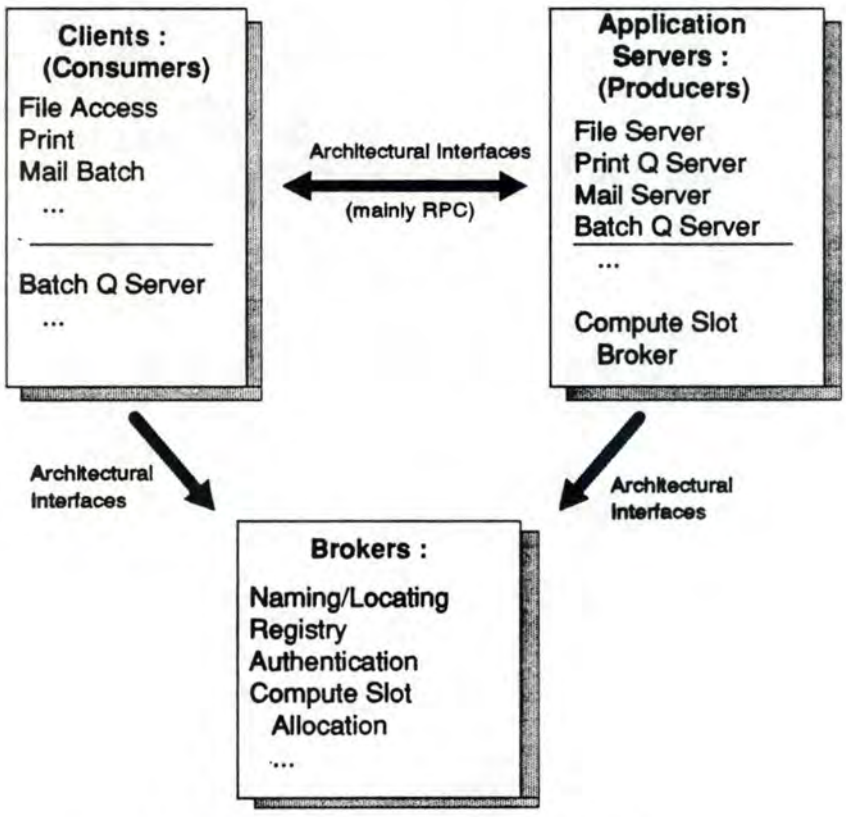

Schéma 1.2 : Modèle Client/Object/Broker

Le client enquête auprès du broker en vue d'obtenir des informations sur les objets du réseau. Les brokers sont classés en fonction du type d'informations qu'ils fournissent.

61

Le *location broker*, qui occupe une position cruciale dans le NCS, se charge de localiser les objets et interfaces du réseau. Pour obtenir l'emplacement d'une ressource, d'une interface remote ou d'une base de données à consulter, le client s'adresse au location broker. C'est pour l'instant le seul broker fourni avec NCS.

L'*authentication broker* garantit une communication sécurisée entre les clients et les serveurs. Il s'occupe de distribuer des autorisations qui conviennent aux différents clients suite à leur identification.

Le *power broker* a pour travail de répartir équitablement la charge du système distribué entre les processeurs disponibles.

Le *network naming broker* se charge de donner des noms significatifs aux noeuds du réseau ainsi qu'aux directories des utilisateurs et du système.

Toute une série d'autres brokers sont imaginables et peuvent être implémentés successivement. On y trouve notamment des brokers de description, d'allocation ou de nommage de ressources.

1.2.2 Les Server Support Tools

Les *outils de support des serveurs* constituent un ensemble de sous-routines de support du développement d'applications complexes dans un environnement distribué. Ils augmentent ainsi les services fournis par le basic heterogeneous interconnect.

Parmi les outils implémentés l'on trouve :

Data Replication Manager (DRM)
Concurrent Programming Support (CPS)
Process Fault Manager (PFM)

Le DRM est un service permettant la réplication de données. Ses caractéristiques sont d'un côté une haute disponibilité des données et de l'autre côté une cohérence faible des différentes bases de données [NCS_TUT]. Il exporte plusieurs interfaces qui permettent aux clients de disposer de répliques des données critiques en plusieurs endroits du système et de gérer celles-ci. Le schéma 1.3 illustre le fonctionnement du DRM et ses interfaces.

A cet instant la seule application faisant usage du DRM à travers l'interface "*drm.imp.idl*" est le *Global Location Broker*. Au point 7.3.2 le lecteur trouvera un exemple d'usage pratique du mécanisme de réplication. L'outil *drm_admin* permet, quant à lui, un accès direct à l'interface "*rdrm.idl*".

62

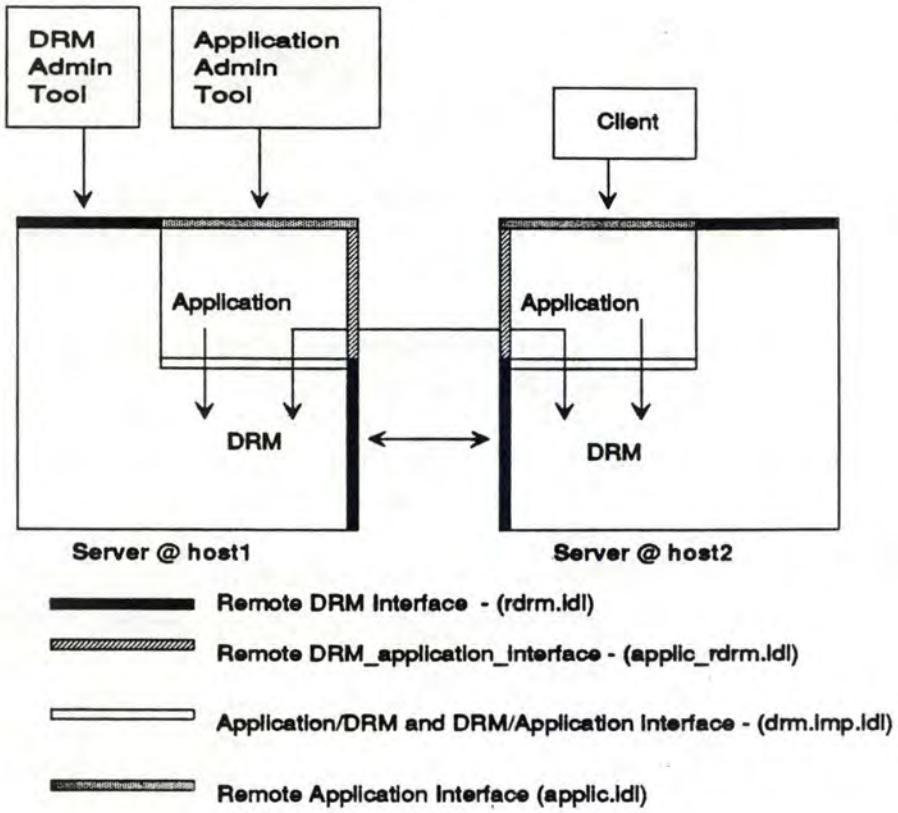

Schéma 1.3 : Les interfaces du Data Replication Manager

Le CPS offre une interface qui permet de créer un environnement multi-tâche à l'intérieur d'un seul processus serveur [CPS]. Dans l'espace d'adressage d'un processus pourront alors se dérouler plusieurs *threads*, processus légers. Le CPS évite l'excès CPU qu'on rencontre dans les systèmes UNIX pour le changement de contexte entre processus et la leur communication. Le CPS ne connaît pour l'instant que quelques implémentations.

De plus le programmeur dispose du *Process Fault Manager* (PFM) qui normalise la gestion de signaux, de fautes et d'exceptions par l'installation de *cleanup handlers*. Ces handlers sont de petits sous-programmes de gestion d'erreurs. Lors de la survenance d'un événement critique, le handler s'occupe de libérer les ressources acquises par l'application avant de la terminer. D'autres types d'erreur peuvent engendrer des messages d'avertissement ou lancer des procédures de correction de la situation fautive.

Les annexes A.2 et A.5 fournissent une description détaillée des interfaces PFM et DRM.

D'autres outils fournissant, par exemple, des services d'accès sécurisé aux ressources ou des mécanismes de transactions atomiques se trouvent encore sous étude.

63

1.2.3 Le Heterogenous Interconnect.

On y trouve les éléments standardisés suivants :

1. Le Remote Procedure Call (NCA/RPC), extension de l'appel procédural classique à un environnement distribué. Le *RPC_RunTime* reprend la librairie des fonctions liés au RPC.

2. Le langage de définition des interfaces des procédures remotes, le *Network Interface Definition Language* (NIDL).

3. Le protocole de représentation des données, *Network Data Representation* (NDR), qui définit les conversions des données structurées en flux binaires afin de les adapter à la transmission dans les paquets RPC.

Ces éléments font l'objet de descriptions plus détaillés dans la suite de ce chapitre.

1.3 NCS et OSF

1.3.1 L'Open Software Foundation

L'OSF, *Open Software Foundation*, est un regroupement de constructeurs informatiques qui s'est donné comme but de briser le monopole du marché UNIX détenu en partie majeure par AT&T et SUN. L'objectif est de créer un système d'exploitation ouvert, portable et performant. Celui-ci va intégrer, suite à des négociations et conférences de standardisation, les meilleures technologies du marché. Le procédé est toujours le même. L'OSF lance un appel d'offres technologiques (*Request For Technologies*, RFT) et tous les producteurs de hardware et de software peuvent y proposer leur technologie. Lors de conférences chacun peut défendre son point de vue et essayer de faire accepter sa technologie. Un comité de standardisation décidera dès lors de la technologie à adopter, qui selon les besoins peut être le recoupement de plusieurs propositions. Par étapes successives, appelées *snapshots*, le système d'exploitation sera fourni d'abord aux membres de l'OSF en vue de le porter sur leurs systèmes et de le vendre ensuite aux utilisateurs finaux.

1.3.2 Le système d'exploitation OSF/1

En automne 1991 le premier snapshot, OSF/1, a été publié. Il est composé de deux sous-ensembles : le *Kernel* et le *Distributed Computing Environment* (DCE). [DCE_ADG] [DCE_PW]

Le Kernel "Mach", proposé par l'université de Carnegie Mellon, dispose d'une mécanisme de communication de processus très performants et de processus légers (*threads*) [Sansom].

Le DCE comprend plusieurs services nécessaires dans le cadre d'un système distribué. Citons-en quelques-uns.

- *Un service de RPC.* Il s'agit du NCS d'APOLLO/HP, qui permet le développement d'applications distribuées et la conversion d'anciennes applications centralisées.

- *Un service d' horloge (time service).* Le "*dts*" de DEC s'occupe de synchroniser les horloges des différents hôtes du système distribué.

- *Un service de directory.* Il s'occupe de gérer les ressources hardware et software du réseau et de fournir les informations pertinentes à ses clients.

- *Un service de fichiers distribué.* Ce service permet la distribution de fichiers en gardant un espace de nommage unique à travers tout le réseau.

2. Les concepts de base

2.1 L'orientation objet

Les architectures RPC classiques permettent l'accès aux ressources en s'adressant directement aux serveurs concernés. NCS a adopté une autre approche, celle de l'*orientation objet.*

Toute ressource du système, qu'elle soit de nature physique ou logicielle, est assimilée à un objet. Les programmes accèdent aux objets à travers des interfaces et se caractérisent par les objets qu'ils manipulent plutôt que par les machines avec lesquelles ils communiquent. Ces programmes *orientés objet* sont plus faciles à mettre au point et s'adaptent sans problèmes aux changements éventuelles des machines ou de la structure des réseaux. L'approche objet dispose de nombreux avantages. L'objet est l'unité

- **d'abstraction** : l'objet sait comment l'opération sera exécutée

- **d'extension** : l'on peut suivre l'évolution du système en créant de nouveaux objets exportant les anciennes interfaces (principe de l'héritage) ou en modifiant uniquement l'interface des objets existant pour refléter les fonctionalités supplémentaires

- **d'intégration** : puisque les objets peuvent être incorporés, les applications développées séparément seront intégrables

- **de distribution** : ils garantissent la transparence de localisation

- **de reconfiguration** : les objets peuvent être migrés sur d'autres machine en vue de l'équilibrage de la charge ou suite à la survenance de pannes

- **de fiabilité** : la réplication d'objets permet une meilleures fiabilité et disponibilité du système

65

2.2 Objets, Types, Interfaces

Un *objet* est une entité accessible via des opérations bien définies. Parmi les objets l'on trouve des fichiers, des directories, des bases de données, des lignes de communications, des imprimantes ainsi que des processeurs.

Tout objet est caractérisé par un certain *type*. Les programmes accèdent à un objet d'un certain type à travers son *interface*. L'interface décrit un ensemble d'opérations appliquables au type d'objet en question. Chaque opération est décrite par ses entrées et sorties en négligeant expressément ses aspects d'implémentation.

Prenons par exemple plusieurs instances d'un objet du type *"file d'attente d'impression"*. Chacun des ces objets est accessible à travers la même interface de *gestion d'impression* qui inclut des opérations d'ajout, de retrait, de modification de priorités ainsi que le listing des travaux introduits dans les différentes files.

2.3 Universal Unique Identifier, UUID

NCS identifie tout objet, type et interface par un identifiant unique universel (*Unique Universal Identifier*, UUID). Il s'agit d'un identifiant à longueur fixe (16 octets) garanti univoque pour l'élément auquel il se réfère. Il est constitué par le compostage de l'identifiant du système (adresse réseau + identifiant de la suite de protocoles utilisée) qui l'a crée ainsi que son temps système actuel.

Les avantages des UUID comme identifiants de bas niveau, au lieu de chaînes de caractères, sont multiples. Les UUID sont facilement intégrables dans des structures de données et souvent de taille inférieure aux strings. Si le besoin se présente il sera possible d'ajouter différents mécanismes de nommage au dessus du concept d'UUID. Finalement les UUID sont crées sans intervention quelconque de serveurs spéciaux ou de représentants humains qui se chargent de distribuer les identifiants respectifs. Un outil spécial de génération d'UUID (*uuid_$gen*) est disponible aux développeurs sur tout noeud du système.

2.4 Les client et les serveurs

Le NCS identifie deux sortes d'agents qui interagissent en vue d'utiliser les ressources limitées du système. Il s'agit des clients et des serveurs.

Le *client* est le programme qui fait des RPC. Un RPC constitue un requête d'exécution d'une opération particulière sur un objet spécifique.

Le *serveur* est un programme qui implémente une ou plusieurs interfaces et qui fournit ainsi l'accès à un ou plusieurs objets. Lorsque le serveur reçoit une requête d'un client il exécute une procédure *manager* qui agit directement sur l'objet et qui fournit éventuellement des résultats à renvoyer au client.

Le schéma 2.1 nous montre l'exportation de trois interfaces par un serveur pour deux types d'objet (*mailing lists et printer queues*).

66

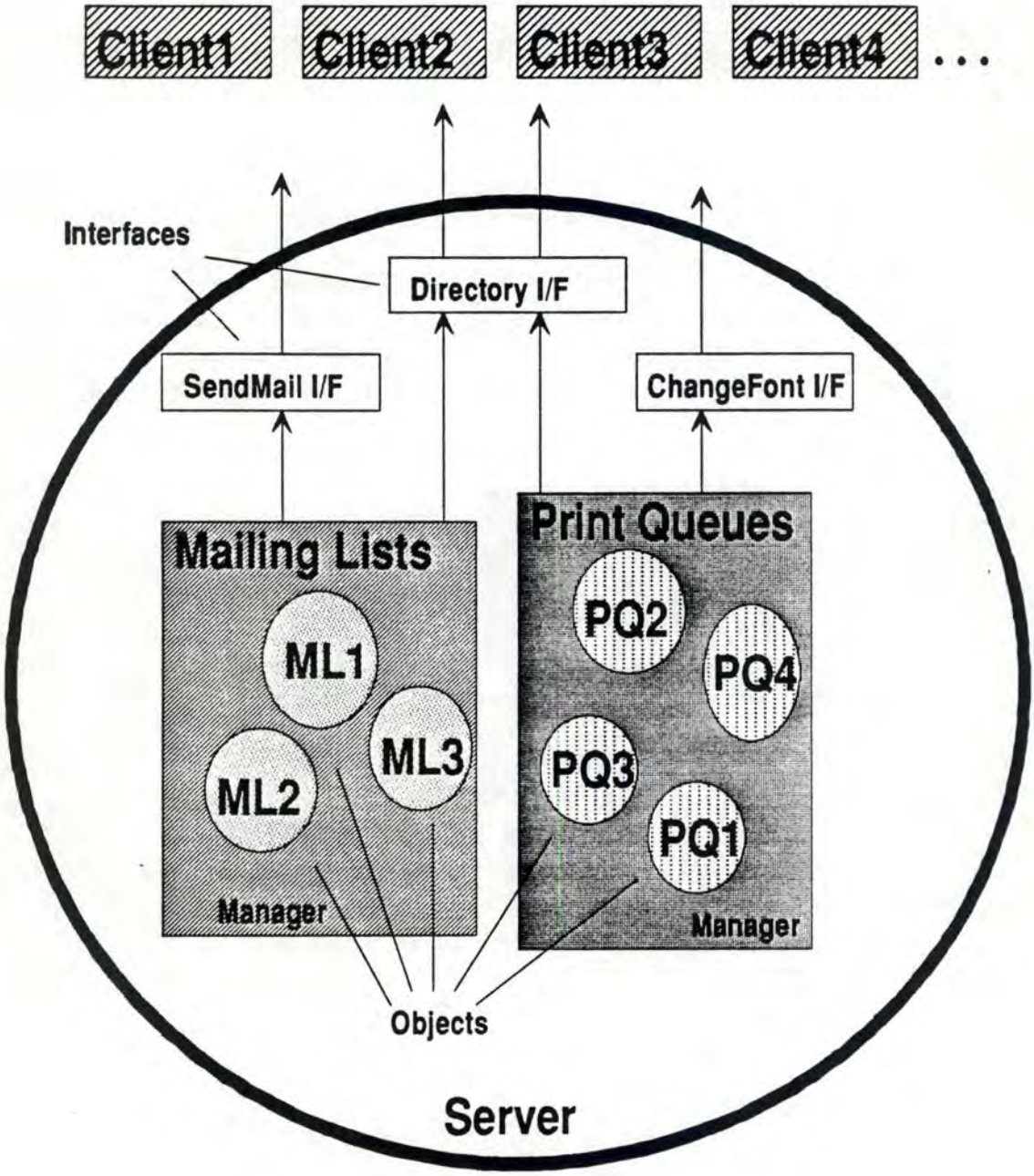

Schéma 2.1 : Objets, Types, Interfaces

3. Le modèle réseau du NCA

NCA utilise un modèle réseau construit sur l'abstraction des *sockets* Berkeley et assume l'existence d'un service de transport du type datagramme au niveau transport.

67

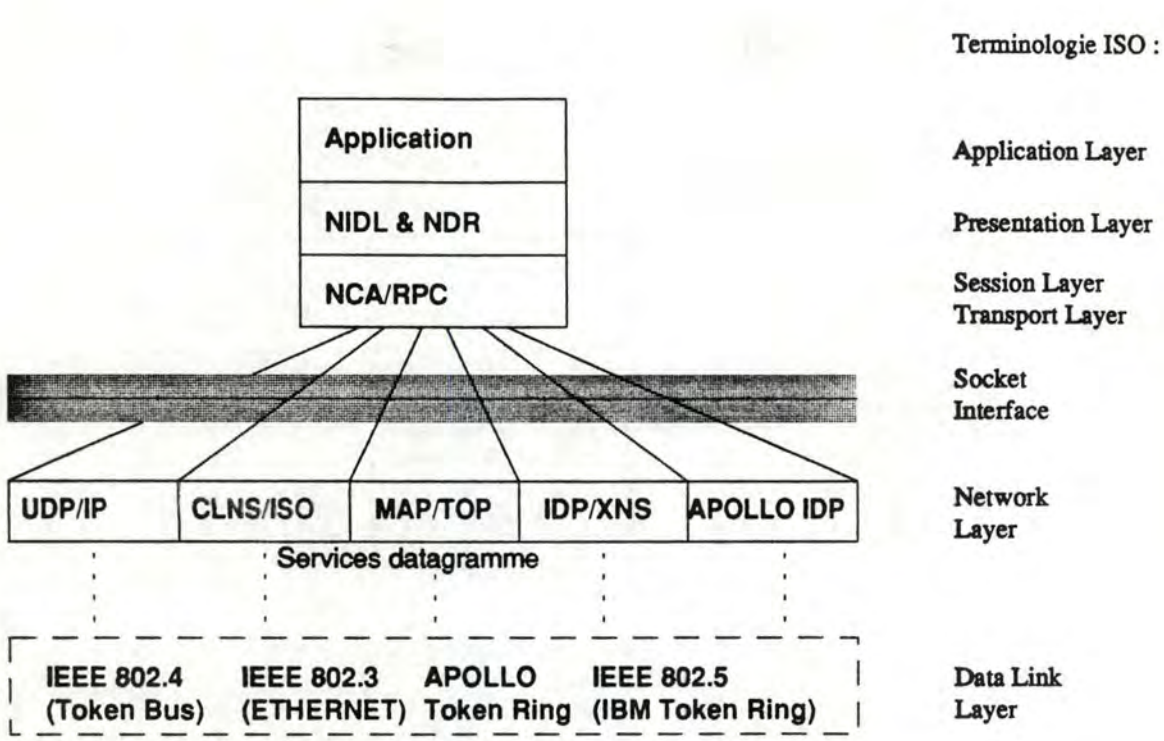

Schéma 3.1 : Modèle réseau du NCA

3.1 Le concept de socket

Une famille de protocoles (*protocol/address family*) est une suite de protocoles de communication liés. La famille de protocoles IP du DoD, par exemple, comprend le TCP (*Transmission Control Protocol*) et l'UDP (*User Datagram Protocol*). [Tanenbaum]

NCA est largement indépendant de la famille de protocoles utilisée. Cette transparence est réalisée par le concept de *socket* comme support de communication. L'on peut considérer le *socket* comme point terminal d'un tuyau de communication. Le serveur scrute un *socket* pour déceler l'arrive de requêtes d'un client. Le client transmet ses messages au *socket* du serveur. Sur le schéma 3.3 le serveur 1 écoute au socket "dds:464a.590f, port 67" et le client 3 y envoie ses messages RPC.

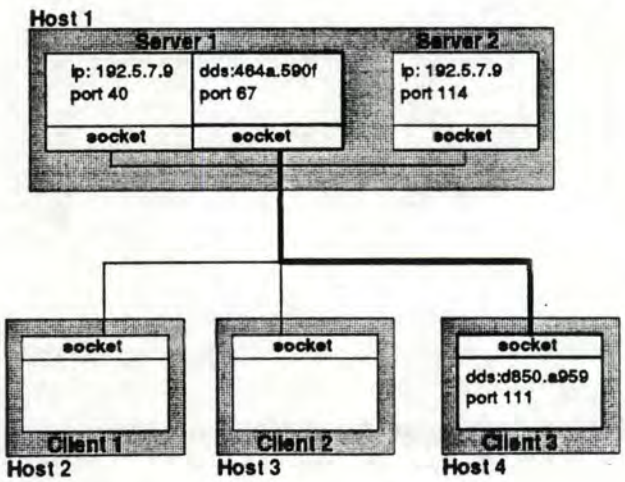

Schéma 3.2 : Communication RPC par sockets

68

Tout socket est univoquement défini par son *adresse socket*, qui contient un identifiant de famille de protocoles, une adresse réseau et un numéro de port.

DDS Socket Address (APOLLO Domain)

Family	Port	Network Address	
16-bit integer	16-bit integer	Network ID	Host ID
		32-bit integer	32-bit integer

IP Socket Address (DARPA Internet)

Family	Port	Network Address
16-bit integer	16-bit integer	Host ID
		32-bit integer

Schéma 3.3 : Structure des adresses IP et DDS

L'*identifiant de famille* est attribué par l'université de Berkeley en Californie.

L'*adresse réseau* identifie, selon la famille de protocoles, une machine particulière, un hôte, à l'intérieur du réseau. Le *port* fait référence à une file de messages d'un socket à l'intérieur d'une machine spécifique. L'on distingue deux types de ports : les *well-known ports* et les *opaque ports*.

Un port est considéré comme *well-known* du moment qu'il est statique, c'est-à-dire compris dans la définition de l'interface remote. Les clients y enverront leur requêtes et le serveur n'écoutera qu'au port spécifié. L'attribution de ces ports se fait en principe par les administrateurs du protocole. C'est ainsi que l'***Internet Protocol*** a destiné le port 23 au service de login remote TELNET. Or, le nombre de ports d'une machine se trouve limité et des risques de conflit entre applications faisant usage d'un même port ne sont guère exclus. Cela nous mène vers l'idée de ports *opaques*, inconnus des clients et des serveurs. C'est une autre instance, le *location broker*, qui s'occupe d'attribuer dynamiquement les numéros de port. Notons que le *location broker*, lui, dispose néanmoins d'un port public pour écouter aux requêtes de ses clients.

3.2 Le service de transport

L'environnement d'un réseau est souvent non fiable et ses failles sont difficilement prévisibles. Le but du protocole RPC est de cacher ces inconvénients et de rendre les appels lointains identiques aux appels locaux.

Une solution serait de prendre un protocole de transport fiable, par exemple, le TCP/IP ou la couche transport du modèle OSI. L'alternative serait d'utiliser un service de transport non-fiable par transmission de datagrammes. Le protocole RPC s'occuperait alors de détecter les erreurs résiduelles.

69

NCA utilise cette deuxième approche pour 3 raisons.

1. La minimisation de l'échange de messages et des autres excédents. En effet les protocoles du genre TCP sont lourds à implémenter et les coûts de l'ouverture et de la maintenance des circuits virtuels sont considérables. TCP nécessite l'échange de huit messages pour l'envoi d'un seul RPC. Des serveurs utilisant TCP sont aussi censés garder des informations d'état sur tous ses clients.

2. La possibilité de prédire la sémantique des RPC. Tandis que la sémantique des RPC est prévisible pour un même protocole de connexion, il en est tout autre lorsqu'on emploi des implémentations et des familles différentes. Certaines fonctionalités nécessaires au RPC peuvent varier d'un protocole à l'autre ou même ne pas exister du tout. Le protocole orienté connexion, sera-t-il capabale de détecter des coupures, d'accepter des données "*out of band*" ou de gérer différents streams binaires ou structurés ? Les protocoles non-connectés sont souvent plus simples et disposent de nombreuses fonctionalités communes à travers toutes les familles existantes.

3. La disponibilité du service. Il est possible que certaines machines, pour des raisons diverses, ne supportent pas de service transport fiable. NCA assume l'existence d'un service de datagrammes, ce qui est pratiquement toujours le cas.

3.3 Les paquets NCA/RPC

Les paquets échangés lors d'un RPC sont constitués d'une entête et d'un corps. L'entête comprend des informations de contrôle du protocole tandis que le corps contient les arguments en entrée et en sortie. Le NCA permet une taille maximale de 80 octets pour l'entête et de 65535 octets pour le corps. Ces valeurs sont souvent sujets aux limitations imposées par la couche transport utilisée. Si la longueur du paquet NCA excède les limites prescrites du logiciel de communication, le protocole NCA/RPC se charge de le décomposer en fragments de taille conforme.

3.3.1 Le format d'un paquet

La représentation binaire de l'entête est déterminée par le type de paquet, par la façon dont ce type sera représenté en NDR et par le format de représentation local des données. Examinons dès à présent la structure en C de l'entête d'un message NCA/RPC.

70

```
typedef struct {
        unsigned small      rpc_vers,
        unsigned small      ptype,
        unsigned small      flag,
        byte                pad1,
        nca_rpc_$drep_t     drep,
        uuid_$t             object,
        uuid_$t             if_id,
        uuid_$t             actuid,
        unsigned long       server_boot,
        unsigned long       if_vers,
        unsigned long       seq,
        unsigned short      opnum,
        unsigned short      ihint,
        unsigned short      ahint,
        unsigned short      len,
        unsigned short      fragnum,
        byte                pad2
    } nca_rpc_$pkt_hdr_t;
```

Champ	Description
rpc_vers	Numéro de la version du protocole. Il sert à distinguer les différentes versions du protocole NCA/RPC qui peuvent exister en parallèle dans un environnement distribué.
ptype	Type de paquet. Il identifie le type de requête effectuée par le client ou le type de réponse renvoyée par le serveur.
flag	Drapeaux. Ces drapeaux binaires servent au support du protocole RPC.
drep	Identifiant de la représentation des données utilisée. Ce champ décrit le format dans lequel se trouvent les différent types transmis : les entiers, caractères et nombres en virgule flottante. Il précise notamment le format des entiers dans l'entête même du paquet. En fonction de la représentation du destinataire celui-ci devra convertir l'entête avant de l'évaluer.
object	Identifiant d'objet. Il indique l'objet auquel se réfère le message RPC. Si aucun objet spécifique n'est concerné ce champ contient la valeur 0 . (uuid_$nil)
if_id	Identifiant d'interface. Le numéro d'une opération avec l'identifiant de l'interface et de l'objet sont les seules informations dont le "dispatcher" du serveur aura besoin pour décider de l'exécution de la procédure remote choisie.
actuid	Identifiant d'activité. Il identifie l'activité du client qui fait un appel. Le serveur s'en sert comme clé de communication avec ses clients.
server_boot	Heure du dernier démarrage du serveur. Le serveur inclut cette information dans tous les paquets vers ses clients. Il trouve son usage dans le protocole client. (voir 8.1)
if_vers	Identifiant de version d'interface. Il est indispensable lors de l'existence de multiples versions d'une même interface.
seq	Numéro de séquence. Chaque paquet dispose d'un numéro de séquence unique pour tous les paquets d'une même activité. Ce numéro combiné à l'identifiant d'activité spécifie de facon univoque un RPC.
op_num	Numéro d'identification d'une opération dans une interface.
ihint	"Interface hint". Champ facultatif utilisable par le serveur pour optimiser la consultation des ses tables. Une requête RPC recoit en réponse une valeur d'index de la table des interfaces du serveur en vue d'éviter à l'avenir, des recherches coûteuses.
ahint	"Activity hint". Index dans la table de lookup des activités d'un serveur.
len	Longueur du corps du paquet.. Elle ne peut excéder 65535 octets.
fragnum	Numéro de fragment. Si la taille du corps d'un paquet RPC dépasse les limites, NCA procède à la transmission de fragments. Chaque fragment dispose de son identifiant.

71

3.3.2 Les paquets du client

1. Le paquet "***requesf***" est composé d'une entête ainsi que du corps contenant les paramètres en entrée à la procédure remote. Il existent cinq variantes du paquet de requête qui se distinguent par la valeur du champ *flag* du paquet.

Type de requête	Description du type de requête
idempotent	Garantie d'exécution de la procédure appelée. Suite à des messages dupliqués la procédure peut cependant être exécutée plus d'une fois.
at most once	Choix par défaut. Garantie d'au plus une exécution de la procédure demandée.
broadcast	Envoi du paquet à tous les serveurs du réseau local. La taille maximale d'un broadcast est fonction du service datagramme utilisé.
maybe	Aucune garantie quant à la réception ou l'exécution de la requête.
maybe/broadcast	Combinaison des spécifications de maybe et de broadcast.

2. Le paquet "***ping***" sert au client pour obtenir des informations quant à l'existence et à l'état d'un serveur. Puisqu'il n'y a pas d'arguments en entrée il n'est constitué que d'une entête.

3. Le paquet "***ack***". Il est envoyé par le client suite à la réception de la réponse à une requête de type "non idempotent". Ce paquet indique au serveur qu'il peut cesser d'envoyer les réponses au RPC en question. Le lancement d'un nouveau RPC aura le même effet.

4. Le paquet "***fack***". Lors de l'envoi de multi-paquets le client acquitte, par l'envoi du "fack", la réception du dernier fragment reçu. Le serveur peut alors cesser de l'envoyer et transmettre le fragment suivant. Pour inhiber l'envoi de ces acquittements le serveur doit mettre le drapeau *nofack* dans ses paquets.

5. Le paquet "***quit***". Il permet d'annuler le RPC en cours de traitement.

3.3.3 Les paquets du serveur

1. Le paquet "***response***". Le serveur envoie le paquet réponse et cela éventuellement en plusieurs fragments de même numéro de séquence avec des numéros de fragment croissants. Il est composé des paramètres en sortie générés par la procédure exécutée.

2. Le paquet "***working***". Ce paquet répond à une enquête "*ping*" du client et lui signale que le serveur est en train de servir son appel remote. Il est composé que de l'entête.

3. Le paquet "***nocall***". Il s'agit de nouveau d'une réplique au paquet "*ping*". Ce paquet, de même structure que le précédent, indique au client que le serveur ne dispose pas d'informations quant à la réception d'un RPC de sa part. L'obtention de ces message indique souvent au client que son message de requête s'est perdu.

72

4. Le paquet "*reject*". Si le serveur refuse une requête d'un client il lui transmet le paquet "*reject*", qui contient en plus d'une entête de paquet une variable expliquant la raison du rejet. (annexe B.)

5. Le paquet "*fack*". Son utilisation est symétrique au paquet client.

6. Le paquet "*quack*". Ce paquet signale au client la réception du paquet "*quit*" et par conséquent l'accord du serveur d'annuler le RPC concerné.

4. Le paradigme RPC

NCA a adopté le paradigme RPC exposé aux chapitre II.

4.1 L'interface

L'interface d'un module est définie dans le langage NIDL. Elle est déclarée indépendamment de la façon dont l'appel sera effectué et de la façon dont les opérations seront implémentées. Le serveur qui implémente les procédures remote *exporte* cette interface. Elle sera *importée* par les clients qui font les appels aux opérations du module.

4.2 Les clients, les serveurs et les managers

Le processus qui fait des requêtes d'exécution de procédures remotes est appelé *client*. Il s'adresse aux fonctions lointaines à travers l'interface RPC importée précédemment.

Le *serveur* exécute les opérations contenues dans une ou plusieurs interfaces. Ces opérations agissent sur divers types d'objets. Le serveur accepte les requêtes des clients, exécute la procédure demandée et retourne les réponses éventuelles au client. Les deux schémas ci-dessus illustrent le déroulement des opérations et le flux des données pour deux configurations de serveur possibles.

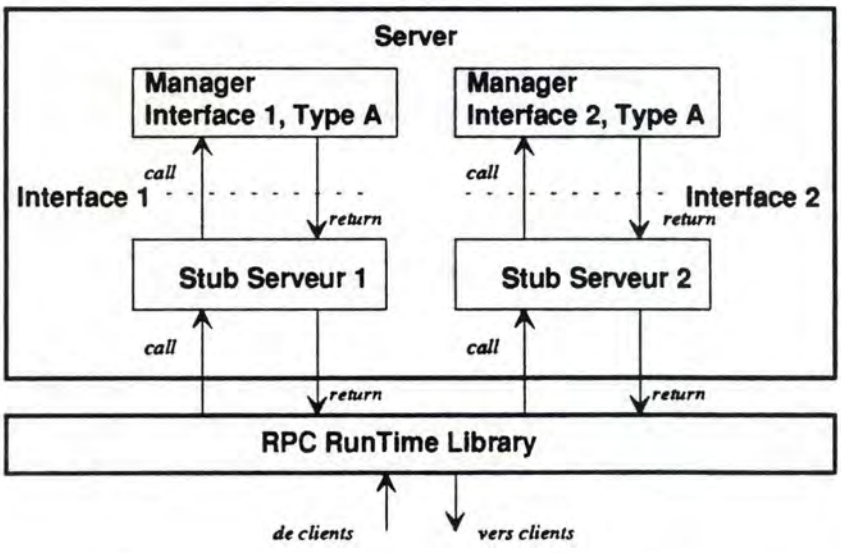

Schéma 4.1 : Serveur exportant deux interfaces

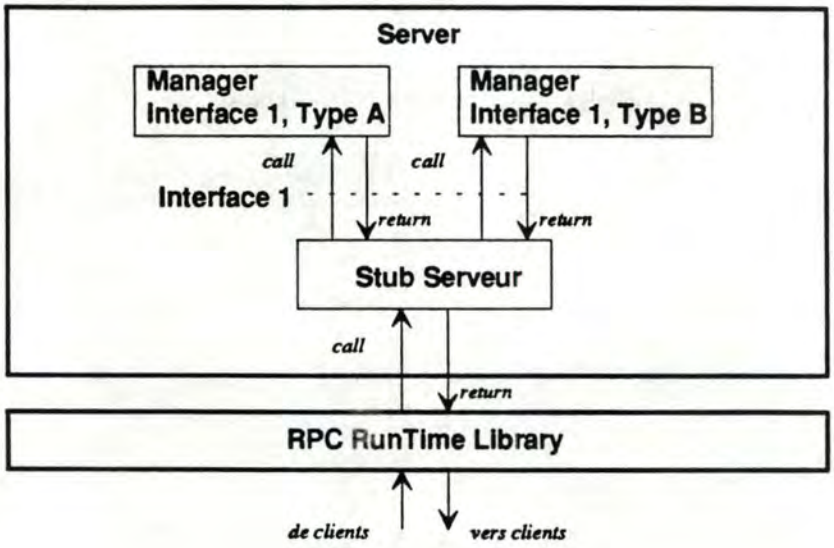

Schéma 4.2 : Serveur exportant une interface pour 2 types

Notons qu'un serveur peut aussi être client, même son propre client. La gestion de répliques d'objets entraîne une étroite collaboration "client-serveur" entre les divers serveurs répliqués.

4.3 Les stubs et le module RPC_RunTime

Les tâches des stubs concordent avec celles définis dans le chapitre II. Les requêtes des clients et les réponses des serveurs passent par les stubs concernés. Le module *RPC_RunTime* implémente le mécanisme RPC en exportant des services de binding, de gestion de handles, d'envoi et de réception de paquets. Son interface est illustrée dans l'annexe A.6 .

4.4 Les handles

En vue de pouvoir envoyer un message de requête de la part du client, le *RPC_RunTime* doit disposer de certaines informations.

1. **L'objet** auquel l'opération demandée se réfère.
2. **L'emplacement du serveur** implémentant les opérations demandées.

Une structure appelée *handle* représente ces informations du côté du client. Un handle ne fait référence qu'à un seul objet. Pendant son existence il peut cependant représenter des liens (*bindings*) vers différents serveurs et même aucun serveur du tout.

4.4.1 Les types de handles

Dans le contexte du NCS un handle n'est rien d'autre qu'un pointeur vers une structure opaque contenant les informations nécessaires en vue d'accéder à un objet spécifique. Le type de handle par défaut est le *handle RPC*. S'il pointe vers une structure autre que celle illustrée au schéma 4.3 l'on parle d'un *handle générique*.

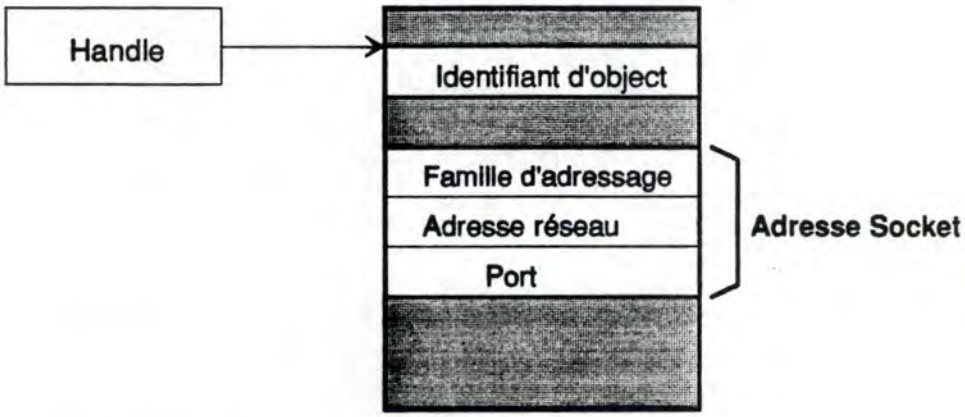

Schéma 4.3 : Représentation d'un handle RPC

4.4.2 Les états de binding des handles RPC

Le tableau 4 expose les 3 états possibles pour un handle RPC.

Etat du binding à l'appel	Information représentée	Mechanisme de livraison	Etat du binding au retour
Non lié	Objet	Broadcast à tous les hôtes du réseau local	Lié complètement
Lié à un hôte	Objet + Hôte	Envoi au "forwarding port" du hôte choisi	Lié complètement
Lié complètement	Objet + Hôte + Port	Envoi au port spécifique de l'hôte	Lié complètement

- **Le handle non lié** (***unbound, allocated handle***) identifie un objet sans pourtant représenter un serveur particulier. Un appel avec ce handle résulte dans la transmission d'un appel broadcast à tous les serveurs à l'écoute. Un serveur exportant l'interface demandée pour l'objet en question peut répondre. La première réponse obtenue servira au client pour mettre à jour sa représentation locale du serveur et de lier le handle.

- **Le handle lié à un hôte** (***bound-to-host handle***) contient l'identifiant d'un objet et d'une machine sans indication du numéro de port du serveur concerné. Si la définition de l'interface comprend un port spécifique, la requête y sera directement envoyée. Sinon elle sera transmise au *Local Location Broker* (**LLB**) du hôte indiqué. Celui-ci connaît tous les serveurs locaux ainsi que leurs interfaces et objets gérés et renvoit les informations manquantes au client.

- **La handle complètement lié** (***bound handle***) contient, en plus de l'UUID d'un objet, l'adresse complète d'un serveur. Ce type de handle permet au RPC_RunTime d'envoyer les messages directement au socket spécifié.

75

L'obtention de la réponse du serveur ou du LLB a comme conséquence le binding complet du handle *non-lié* ou *lié-hôte*. Pour les appels qui suivent il ne sera alors plus nécessaire de refaire le binding.

4.5 Les handles et le binding

NCS offre deux types de binding et deux représentations possibles de handles.

	Binding manuel	**Binding automatique**
Handle explicite	Type : handle_t Représentation : paramètre	Type : générique Représentation : paramètre
Handle implicite	Type : handle_t Représentation : variable globale du client	Type : générique Représentation : variable globale du client

4.5.1 Les handles implicites et explicites.

Un handle *explicite* est passé comme premier argument de chaque RPC. Il est transféré explicitement du premier appel effectué par le client jusqu'au stub du serveur. Le RPC_*RunTime* du client y insère l'adresse du client lors de l'appel, ce qui permet au serveur de connaître la destination des réponses.

Du moment que le handle est représenté comme une variable globale l'on parle d'un handle *implicite*. Comme l'opération appelée ne comprend plus de handle en paramètre, le serveur n'y aura plus accès. Au prix d'une meilleure transparence il introduit certaines limitations :

- Sans handle le serveur client ne peut opérer que sur un seul objet, à moins qu'il passe en argument à la procédure un autre identificateur d'objet.

- Comme toutes les opérations d'une interface se partagent la même variable globale, le client ne saura accéder qu'à un seul serveur à la fois. La division d'applications en plusieurs parties s'exécutant parallèlement devient compliqué avec ce genre de handle.

76

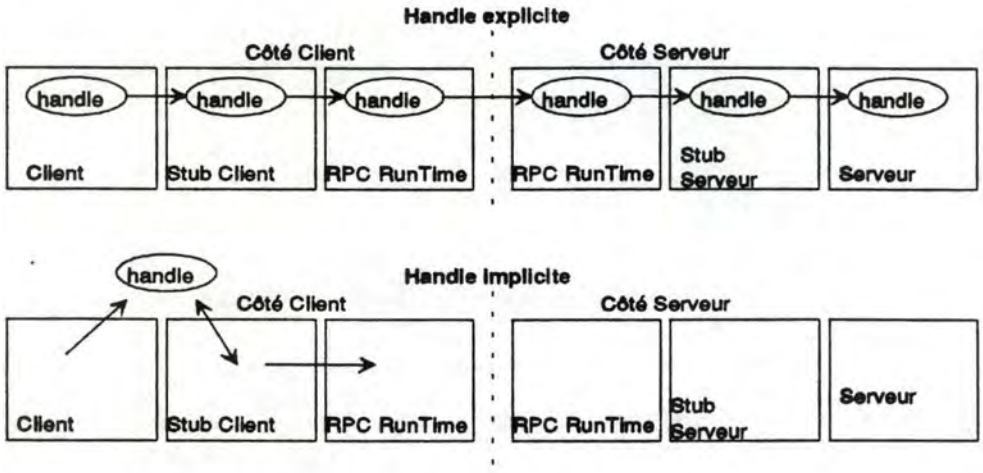

Schéma 4.4 : Handles implicites et explicites

4.5.2 Le binding manuel et automatique

Le *binding manuel* implique l'emploi de *handles RPC*. En vue de créer et de lier les handles le client s'adresse directement aux opérations du RPC_*RunTime*.

L'usage de *handles génériques* implique un *binding automatique*. Le développeur est censé fournir des routines de *binding* et de *unbinding automatiques*, procédures traductrices du format générique vers un format compréhensible par le RPC_RunTime et inversément.

Il est clair que la flexibilité du binding automatique ne va pas sans sacrifices du côté performance. Le procédé automatique requiert des traductions à l'intérieur des stubs pour chaque RPC. Le déroulement de ces deux sortes de binding est expliqué dans le tableau ci-dessous.

Binding Manuel	Binding Automatique
1. Client : Générer handle RPC Lier handle RPC selon besoins Appel procédural au stub 2. Stub client: Envoyer requête au serveur Attendre Recevoir réponse du serveur Retourner au client 3. Client : Recevoir résultats du stub client Libérer handle RPC selon besoins	1. Client : Utiliser handle générique pour faire un appel procédural au stub 2. Stub client : Lancer routine de binding automatique 3. Routine d'autobinding Créer handle RPC à partir du handle générique Lier handle RPC selon besoins Retourner handle RPC au stub client 4. Stub client : Envoyer requête au serveur Attendre Recevoir réponse du serveur Retourner au client Appel de la routine d'autounbinding 5. Routine d'autounbinding : Libérer handle RPC selon besoins Retourner au stub client 6. Stub client : Retourner au client 7. Client: Recevoir résultats du stub client

77

5. La définition d'interfaces remotes

Les interfaces des modules remotes sont rédigées dans le langage NIDL (*Network Interface Description Language*). Ces définitions d'interfaces passent par le compilateur NIDL, qui en crée les procédures stub pour le client et les serveur.

5.1 Le langage NIDL

La définition d'une interface décrit les constantes, types de données et les opérations associés à l'interface. Le langage NIDL est purement déclaratif et existe en deux variantes syntaxiques, l'une proche du langage C et l'autre proche du langage PASCAL. Dans la suite nous donnerons une courte introduction à la syntaxe C. L'annexe C. contient la définition complète de la syntaxe NIDL.

Une définition en NIDL a la structure suivante :

identificateur de syntaxe
[*liste d'attributs de l'interface*] **interface** *identifiant*
{
déclarations d'importations
déclarations de constantes
déclarations de types
déclaration des opérations
}

a) **L'identificateur de syntaxe**. NIDL accepte deux syntaxes, l'une est proche du langage C et l'autre ressemble au PASCAL.

b) **L'entête** contient la liste des attributs et l'identifiant de l'interface. La liste des attributs comprend l'UUID, la version et éventuellement le ou les ports publics de l'interface. Elle spécifie aussi si la procédure est locale ou remote ainsi que l'usage de handles implicites et de leur type.

c) **La déclaration d'importation** favorise la modularisation des déclarations, en incluant d'autres textes NIDL dans la définition actuelle.

d) **La déclaration de constantes** associe un nom à une valeur fixe. NIDL interdit les expressions constantes.

e) NIDL permet **la déclaration de types** de données sous la forme:

typedef [*liste des attributs*] *type_spécificateur déclarateurs*.

La *liste des attributs* spécifie si le type est un handle ou s'il n'est pas transmissible dans sa forme actuelle.

Le *spécificateur de type* est un scalaire (byte, nombre entier ou flottant, caractère ...), un type construit (string, tableau, structure, pointeur ...) ou bien un type défini précédemment.

Les *déclarateurs* sont les noms attribués aux nouveaux types. Le déclarateur pour un pointeur est précédé du symbole "*". Celui d'un tableau est suivi des indications de sa taille.

Prenons quelques exemples :

```
typedef long integer32, int32;
typedef int *pointeur_vers_entier;
typedef struct {
   char tab1[32];
   int  toto;
   long resultat;
   } exemple_t;

typedef int tableau_6X4 [6] [4];
```

f) La déclaration des opérations est analogue à l'entête d'une fonction en C. Elle prend la forme générale suivante :

[*attributs_opération*] *type_sortie nom_opération* (*liste_paramètres*) ;

Parmi les **attributs de l'opération** on trouve les mots clé *idempotent*, *broadcast*, *maybe* et *comm_status*. L'attribut *comm_status* spécifie que l'opération appelée est censée fournir en résultat un statut de communication. Ceci permet au stub client de réagir aux erreurs survenues en fonction du résultat obtenu. Les autres attributs indiquent la sémantique RPC à employer par le RPC_RunTime. Par défaut celle-ci est du type "*at most once*".

Le *type_sortie* spécifie le type de données que la fonction retournera. Tout type non-pointeur est admissible et le système renvoit par défaut un entier. Un fonction qui ne retourne pas de valeur est précédée du mot clé *void*. Si l'attribut d'opération *comm_status* est présent le *type_sortie* est obligatoirement *status_$t*. La structure de *status_$t* et les différents messages d'erreur sont exposés dans l'annexe B.

Le nom de l'opération identifie les différentes opérations de l'interface.

La liste des paramètres contient les paramètres séparés par des virgules. Elle se structure comme suit :

param_type [*field_attribute_list param_attribute list*] *nom_paramètre*;

La liste des attributs des champs s'applique aux tableaux et comprend les mots clé *last_is* et *max_is* suivi d'un nom de variable indiquent dynamiquent le dernier index ainsi que l'index maximal d'un tableau à passer lors du RPC.

La liste des attributs des paramètres stipule si un paramètre est en entrée ou en sortie, ou même les deux à la fois, par les mots clé *in* et *out*. Un attribut spécial *comm_status* souligne le fait qu'on est en présence d'un attribut de statut.

79

Le nom du paramètre doit être précédé de "*" pour désigner les paramètres *in* et *out*, sauf les tableaux, passés par référence.

Une étude plus détaillée de la syntaxe C ou PASCAL du langage NIDL dépasserait le cadre de ce mémoire. Le lecteur intéressé est renvoyé à la lecture des chapitres 6,7 et 8 du manuel de référence du NCS.

5.2 Le compilateur NIDL

Le compilateur NIDL crée à partir de déclarations d'interfaces les modules *stub* qui seront linkés avec les clients et les serveurs. Il génère, selon la volonté de l'utilisateur, du code source en C ou en PASCAL. Le linkage avec des modules est aussi prévu par le NCS.

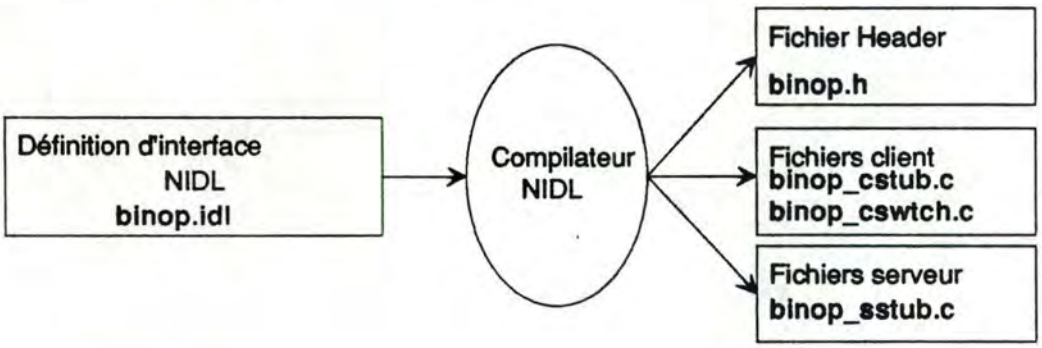

Schéma 5.1 : Le compilateur NIDL, input et output

En plus de ces stubs le développeur obtiendra un fichier header. Ce fichier initialise les structures de gestion du NCS et contient les déclarations des fonctions RPC_RunTime, des constantes et types de support des appels remotes. Il initialise la structure *if_$spec* que le serveur passe au RPC_RunTime lors de l'exportation de ses interfaces. La structure *epv_t* représente une entrée, un pointeur vers une opération dans la table des procédures manager (*entry point vector*, EPV).

Les fichiers _cstub.c et _cswtch.c forment le *stub client* tandis que le fichier _sstub.c constitue le *stub* associé au *serveur*.

Le fichier *switch* (_cswtch.c) joue un rôle spécial dans le cas de serveurs répliqués. Un serveur répliqué fait des RPC à ses répliques, à travers les fonctions compris dans ses stubs. Il exécute aussi ses procédures managers suite aux demandes de ses clients. Dans ce cas le développeur risque d'assister, lors de la compilation, à des conflits de dénomination : Comment distinguer entre les appels aux managers locaux et les procédures remotes du stub serveur ?

Ce problème est résolu par le fichier *switch*. Il contient des procédures "publiques" au contraire du fichier *stub* qui lui ne contient que des déclarations de procédures "locales" et invisibles en dehors de son code source. Le *switch* accède aux fonctions du *stub* par le biais de son *EPV*.

80

Un client ordinaire est linké avec ces deux fichiers. Tout appel public sera relayé au stub respectif. Un serveur répliqué n'est linké qu'avec le stub client. Les requêtes remotes du serveur s'adresseront alors directement au EPV tandis que les noms "publics" feront référence aux managers locaux.

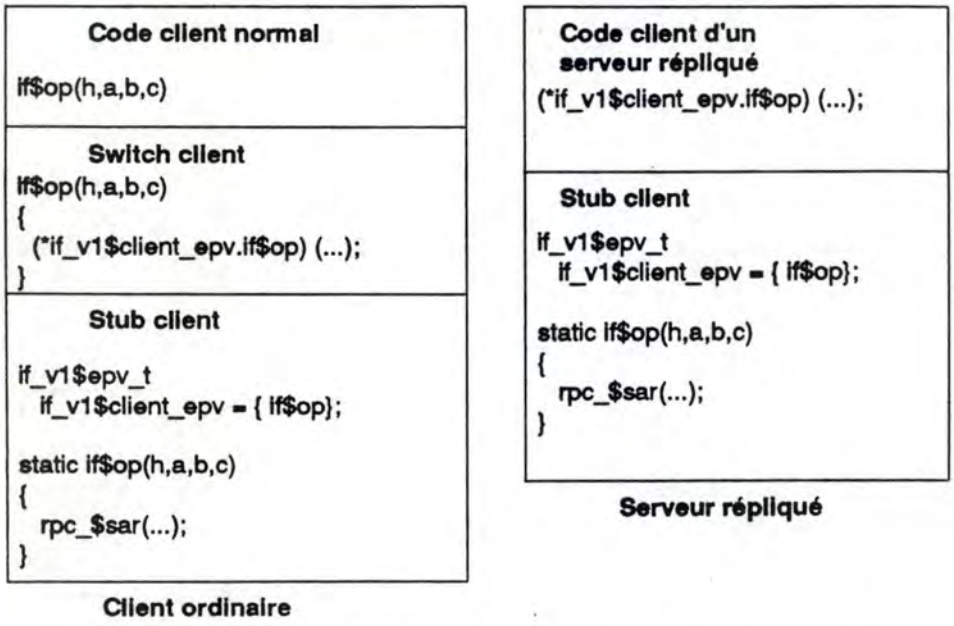

Schéma 5.2 : Relations entre les clients,switch et stub.

5.3 Aspects avancés du NIDL

5.3.1 Les tableaux *ouverts.*

NCS prévoit la transmission de tableaux de taille dynamique. Voici un exemple appliquant les attributs *last_is* et *max_is*.

```
typedef struct {
        int pmax;  /* index maximal */
        int plast; /* index du dernier élément */
        long [max_is(pmax), last_is(plast)] parray[];
        }
```

5.3.2 La conversion de types de données.

Une structure de données présente (*PRES*) trop complexe aux procédures de linéarisation des stubs est automatiquement traduite en une structure transmissible (*TRANS*) par l'emploi du mot clé *transmit_as*. Le programmeur devra cependant mettre au point les procédures de traduction selon les prototypes suivants.

1. void *PRES*_to_xmit_rep()

Elle effectue la conversion du type présenté vers le type transmissible après en avoir alloué de la mémoire.

81

2. **void** *PRES*_**from_xmit_rep**() est l'opération symétrique de 1.

3. **void** *PRES*_**free**() libère la mémoire allouée pour le type présenté en 2.

4. **void** *PRES*_**free_xmit_rep**() libère la mémoire allouée pour le type transmissible en 1.

La conversion de types a une influence sur les performances des RPC. Le développeur peut, par exemple, décider de ne transmettre que les informations utiles d'un tableau peu rempli (*sparse array*).

5.3.3 Le binding automatique.

La traduction de/vers les handles RPC et les handles génériques (*GENERIC*), est réalisé par les procédures suivantes.

1. **handle_t** *GENERIC*_**bind** () génère un handle RPC à partir d'un handle générique et le renvoie en résultat.

2. **void** *GENERIC*_**unbind**() brise le lien établi entre les deux handles et libère toutes les ressources allouées en 1.

5.3.4 L'existence de multiples interfaces.

Le langage NIDL permet l'exportation de plusieurs versions d'une même interface par un serveur.

Sachant que l'UUID des différentes versions est identique il restera au client de déterminer la version appropriée en comparant les numéros des versions exportées. Notons ici l'existence de l'opération *rrpc_$inq_interfaces* qui donne en résultat le vecteur des interfaces exportées par un certain serveur.

Le serveur, quant à lui, déclare un vecteur d'entrée pour chaque version exportée séparément. Si l'implémentation des opérations connaît aussi des mutations le programmeur devra en tenir compte dans son module manager et attribuer des noms uniques aux différentes opérations.

5.3.5 L'existence de plusieurs managers

NIDL permet l'exportation de procédures manager pour plusieurs types d'objets. Restera cependant au client de trouver l'interface convenable et au serveur de prévoir des vecteurs d'entrée ainsi que des modules managers différents.

82

6. Network Data Representation

NDR spécifie comment les types de données structurés manipulés par les applications et le compilateur NIDL seront encodés en flux binaires avant leur transmission à travers un support de communication. On y spécifie aussi une représentation des données permettant l'interopérabilité de machines hétérogènes. En ce sens NDR remplit la fonction de la *couche de présentation* du modèle de référence OSI.

Une liste complète des types transmissibles par NDR ainsi qu'une description de leur format de présentation se trouve aux annexes en D.

6.1 Le protocole de présentation des données

NDR définit une série de types scalaires ainsi que de types construits. Le protocole s'occupe de gérer les différents alignements des données en mémoire centrale. Par exemple les données sont alignés sur 16 bits pour le processeurs 680xx et sur 32-bits pour le processeurs VAX. En vue de ces différences il devient clair que des transferts de données risquent de devenir coûteux et dans certains cas même impossibles.

Pour cette raison NDR requiert l'*alignement naturel* des données dans les flux binaires. On y aligne toutes les valeurs scalaires de taille inférieure 2^*n* sur des frontières multiples de 2^*n* bytes, où NDR attribue à *n* une valeur maximale de 3. Ceci permet la communication aisée entre machines hétérogènes par l'usage efficace d'opérateurs naturels sans fautes d'alignement. Ceci entraîne cependant l'existence de trous (*gaps*) dans les flux binaires. Il est du ressort du programmeur d'optimiser l'organisation des données afin d'éviter ces *gaps*.

6.2 Le protocole de conversion de données

Les représentations des données varient d'une architecture à un autre. L'ordre des bytes et la taille des types de bases diffèrent souvent. Le *protocole multicanonique* limite les conversion de données en adoptant la philosophie *"receiver makes it right"*. C'est à dire qu'au plus seul le récepteur aura à effectuer une conversion des données en son format local. NDR supporte un ensemble de types de données qui couvre la majorité des représentations courantes sans toutefois posséder le caractère universel d'un approche purement *canonique*. L'on peut supposer que les concepteurs ont accordé une importance plus grande aux aspects de performance qu'à l'universalité de leur système.

Les différentes représentations connus par NDR sont identifiées par un label. Ce *format label* se compose comme suit :

83

Byte 1	Représentation integer (4 bits)	Représentation character (4 bits)
Byte 2	Représentation floating point	
Byte 3	Réservé à un usage futur	
Byte 4	Réservé à un usage futur	

Ce label se trouve dans chaque paquet RPC. Son contenu est établi dynamiquement à chaque RPC avec les valeurs suivantes :

Type de données	Valeur	Format
character	0 1	ASCII EBCDIC
integer	0 1	big-endian little-endian
floating point	0 1 2 3	IEEE VAX Cray IBM

7. Le Location Broker

Un système distribué est composé d'objets de toute sorte, représentations de personnes, de ressources ou de services. Ceux-ci sont de nature temporaire et connaissent une certaine mobilité. Le *location broker*, élément clé du NCS, permet la localisation dynamique et la validation d'existence de ces objets et des interfaces liées.

A la demande des clients, le *location broker* consulte sa base de données et leur renvoie les informations nécessaires.

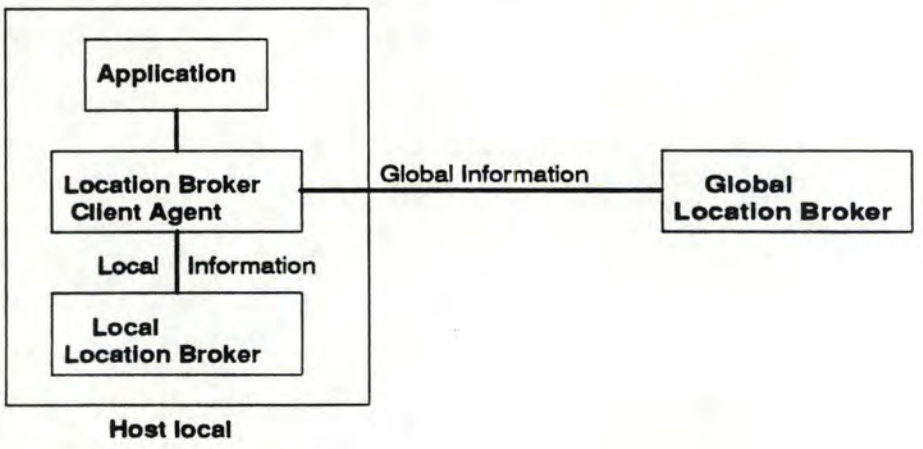

Schéma 7.1 : Structure du Location Broker

84

7.1 Structure du Location Broker

7.1.1 Le Local Location Broker (LLB)

Il s'agit d'un serveur RPC qui maintient une base de données locale contenant des informations sur les objets et interfaces qu'on trouve sur le host local. Il offre des services de *lookup* et de *registering*. Un service spécial, celui du *forwarding* est expliqué en 7.3.3.

7.1.2 Le Global Location Broker (GLB)

Le GLB gère les objets et interfaces disponibles sur tous les hosts du système distribué. Certaines implémentations en font un service répliqué, construit sur le *Data Replication Manager* (DRM).

7.1.3 Le Location Broker Client Agent (LBCA)

Le LBCA est une librairie implémentant les différentes opérations d'accès au LLB et au GLB. Toute opération liée au location broker passe par l'agent du host local.

7.2 La base de données du Location Broker

Une entrée de la base de données se compose comme suit :

Champ	Description
Object UUID	Identifiant unique de l'objet
Type UUID	Identifiant unique du type de l'objet
Interface UUID	Identifiant unique de l'interface
Flag	Drapeau qui indique si l'objet est local ou global
Annotation	Description textuelle de l'entrée courante
Socket Address Length	Longueur du champ suivant : Socket Address
Socket Address	Adresse du serveur exportant l'interface

Il est à remarquer que toute combinaison différente d'objet, de type ou d'interface doit être enregistrée séparément. Pour une liste exhaustive des opérations de recherche et d'enregistrement le lecteur est avisé de consulter A.4 aux annexes.

85

7.3 Utilisation du Location Broker

7.3.1 Enregistrement et recherche

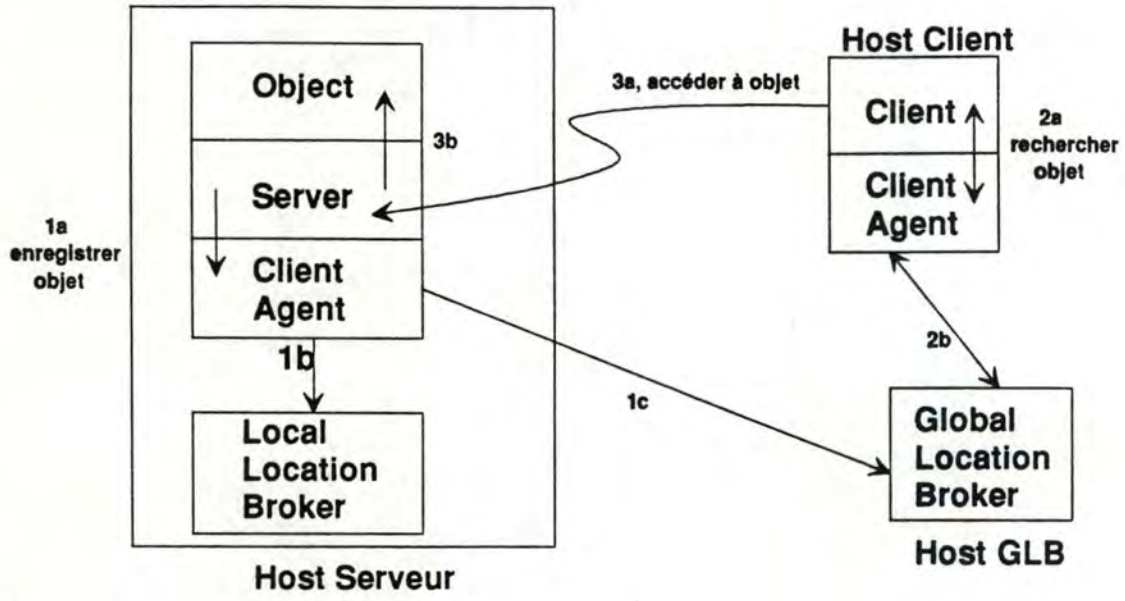

Schéma 7.2 : L'agent client et les location brokers

Ce schéma nous illustre le cas classique d'exportation et d'importation d'interface par le biais du location broker.

La première phase consiste dans l'enregistrement du serveur qui exporte ses services. Le serveur s'adresse au LBCA (1a) pour *s'enregistrer* auprès du LB. Le LBCA enregistre le serveur en question auprès du LLB (1b) et du GLB (1c).

En deuxième phase le client tentera de localiser le serveur mentionné. Il fera un *lookup-call* au LBCA (2a). Le LBCA entrera en communication avec le GLB afin de disposer de l'adresse socket du serveur (2b).

Par la suite le client et le serveur peuvent entrer en communication directe. (3a, 3b)

7.3.2 La réplication du GLB

Sur le schéma 7.3 nous découvrons deux exemplaires du GLB. Chaque GLB répliqué dispose d'une liste de répliques existantes, de sa copie locale des données ainsi que d'une liste de propagation. A chaque appel d'enregistrement ou d'effacement au GLB, le DRM y lie un *timestamp* et le fait propager vers toutes ses répliques. Une propagation terminée, l'entrée du GLB réplique concerné est enlevée de la liste de propagation. Les règles de propagation, assurant la cohérence des différentes bases de données, sont listées ci-dessous.

1) La propagation est uniquement effectuée par la réplique GLB d'origine.
2) Si une copie identique d'une entrée GLB existe déjà pour le même serveur (objet, type, interface et adresse socket sont identiques), le système ne retient que l'entrée ayant le timestamp le plus récent.
3) Si les horloges de deux répliques diffèrent de plus de 10 minutes aucune propagation n'aura lieu

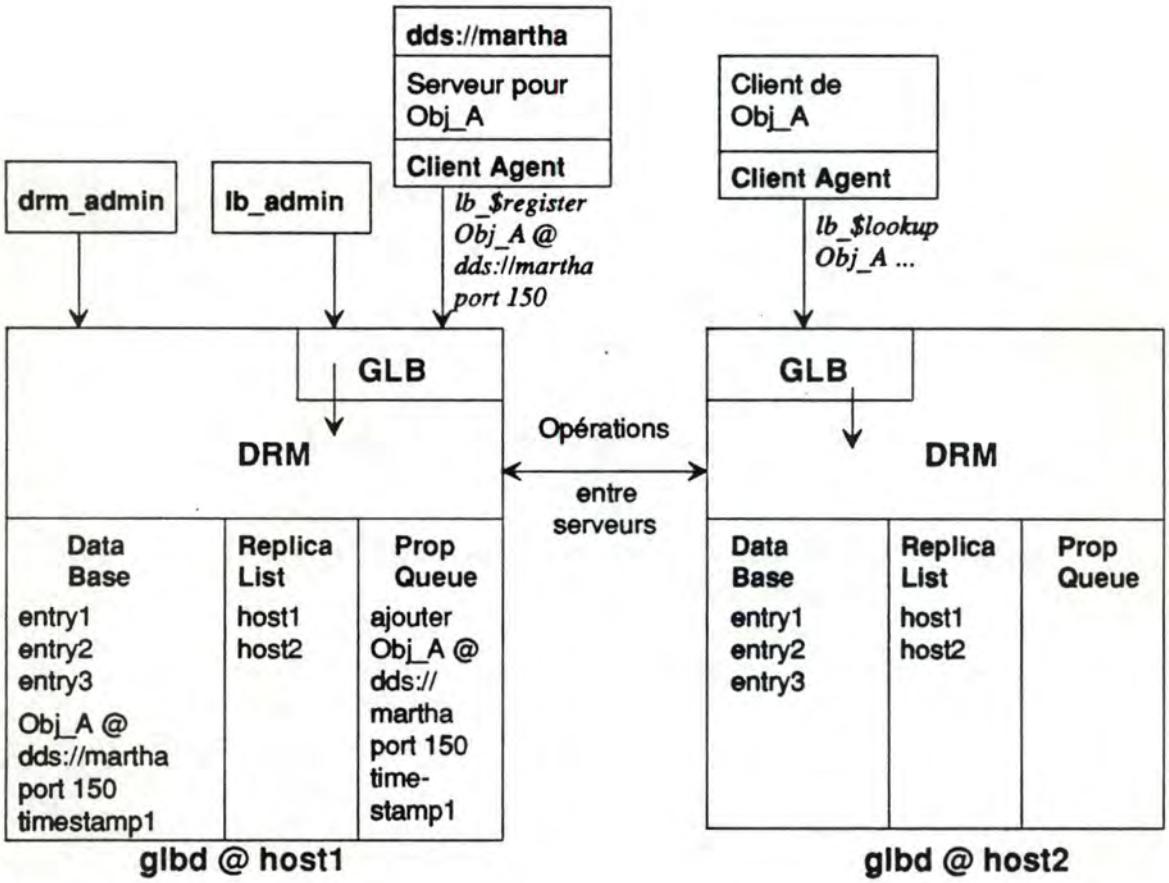

Schéma 7.3 : GLB répliqué, enregistrement et lookup

Si certains GLB sont inaccessibles leurs entrées restent dans la liste de propagation. Le DRM essaie toutes les 15 minutes de recontacter les GLB fautifs et ceci pendant 2 semaines. Après cette date les entrées sont simplement enlevées de la liste. Ceci introduit des incohérence de la base de données répliquée. Les outils administratifs *lb_admin* et *drm_admin* servent à remédier manuellement à cette incohérence.

La phase de *lookup* correspond en principe au cas classique. Notons que l'indisponibilité provisoire d'un GLB force les clients à s'adresser à une de ses répliques.

87

7.3.3 Le forwarding du LLB

Si le client ne dispose pas du port spécifique d'un serveur, mais du host sur lequel celui-ci tourne, il adresse sa requête au *forwarding port* du LLB de ce host. Le LLB, qui reçoit les informations sur l'objet et l'interface locale, se chargera alors de compléter l'adresse socket du serveur en question et de lui relayer le message. Le LLB renverra cette adresse au client afin d'établir pour la suite une communication directe et efficace.

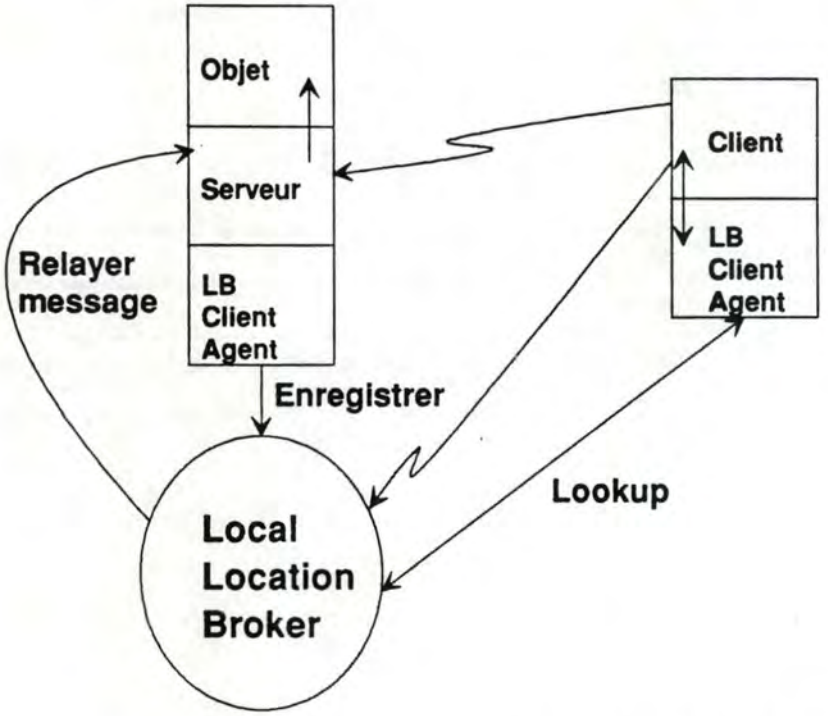

Schéma 7.4 : Utilisation du Local Location Broker

7.4 Les outils administratifs du Location Broker

7.4.1 L'outil *lb_admin*

lb_admin est un outil de gestion des enregistrements de serveurs auprès du location broker (GLB et LLB). Il sert à la consultation, à l'enregistrement et à la mise à jour des bases de données des location brokers. La modification d'un GLB répliqué est évidemment propagée à travers l'entierté de ses répliques.

7.4.2 L'outil *drm_admin*

drm_admin permet de gérer les serveurs basés sur le DRM. Il dispose de fonctionalités d'inspection et de modification de la liste des répliques, de "remise en état de cohérence" des différentes versions de la base de données, d'arrêt de serveurs et d'effacement de répliques.

88

8. Le protocole NCA/RPC

Le protocole RPC *request/response* sera représenté dans les annexes G sous forme de machine à états finis, *finite state machine* (FSM) [NCA][NCA_2]. Une FSM comprend un ensemble d'*états*, d'*inputs* et de *transitions*.

. Une FSM se trouve toujours dans un état courant qui change en fonction des inputs qu'il reçoit.

. Une transition de l'état Si vers Sj est un tuple [*Si, input, condition, Sj, action*]. Une transition est franchissable et son action associée est exécutée si la *condition*, expression booléenne, est vraie.

. Les inputs d'un état, qui sont de quatre types.

Type d'input	Description
message	contenu d'un paquet RPC
timeout	input entrant en action si un état se trouve sans input depuis une certaine durée
notification d'exécution	Notification engendrée par le FSM serveur pour indiquer les changements survenus dans l'exécution de la requête RPC

8.1 Le protocole client.

Le protocole client consiste en 4 FSM.

Nom FSM	Description
SAR	Protocole "Send-Await-Reply". Gestion de requêtes pour l'exécution de procédures "idempotent" et "non-idempotent". (1)
Broascast	Envoi de la requête à tous les serveurs dr réseau local. (2)
Maybe	Envoi de la requête sans attente de la réponse. (2)
Broadcast / Maybe	Combinaison des FSM "Broadcast" et "Maybe"

Remarques :

(1) Le respect de la règle "*at most once*" pour les procédures *non-idempotent* nécessite l'écoute du client aux *requêtes callback* du serveur. Pour cela le client implémente et exporte l'interface du *conversation manager*.

89

En principe le serveur dispose d'informations lui permettant d'éviter l'exécution multiple de procédures. Des exceptions subsistent cependant :

La requête recue est la première du client en question.
La requête est exécutée (!) et toute information à son sujet est abandonée suite à un timeout de confirmation du client.
La requête est exécutée (!), mais le serveur subit un crash avant le renvoi des résultats au client.

Le seul moyen pour distinguer entre un nouveau RPC et un double erronée d'une vieille requête est alors de faire un *callback* au client. Le serveur s'adresse au *conversation manager* du client en lui fournissant l'heure de son dernier démarrage ainsi que le numéro de séquence de la requête "douteuse". Si le numéro du RPC douteux diffère de celui émis actuellement par le client ou si ce client est en état de repos ou même inexistant, la demande est ignorée. La comparaison des heures de *boot* détenus par le client et le serveur permet au serveur d'être au courant de son crash et d'identifier les requêtes doubles.

2) Les requêtes des FSM "broadcast", "maybe" et "broadcast/maybe" s'adressent uniquement à des procédures *idempotent*.

8.2 Le protocole serveur

Le protocole serveur consiste en une FSM qui caractérise la réponse du serveur à un seul client. Le mécanisme de sélection du client et la gestion simultanée [CPS] de plusieurs clients varie en fonction de l'implémentation.

9. Développement d'applications distribuées

Le développement d'applications distribuées sous NCS s'avère être une tâche plutôt facile. Le tableau compare le cycle de développement classique avec celui sous NCS [NCS_TUT] [DCE_ADG].

Développement sans NCS	Développement avec NCS
1) Ecrire application	1) Spécifier en C ou PASCAL la définition des interfaces remotes et les fournir en entrée au compilateur NIDL
2) Déterminer disponibilité des ressources, leur type et leur emplacement	2) Ecrire code de l'application
3) Invoquer procédure lointaine dans le code de l'application	3) Utiliser le "Location broker" pour trouver l'interface de l'objet sur lequel on désire travailler
4) Transférer les données	4) Travail effectué par le RPC_RunTime du NCS
5) Traduction des données par le serveur	5) id.
6) Initier le processus manager concerné	6) id.
7) Valider la fin de l'exécution du manager	7) id.
8) Renvoi des réponses obtenues au client	8) id.
9) Traduction des données par le client	9) id.
10) Terminer invocation de la procédure remote	10) id.

Le développeur suit la démarche classique de développement de logiciels. Il détermine ensuite la distribution des modules serveurs selon des objectifs de performance, de disponibilité de service et de sécurité. Le langage NIDL lui servira pour décrire les interfaces lointaines des serveurs. Le compilateur NIDL fournira des stubs qui seront linkés avec le code source des clients et des serveurs de l'application en question.

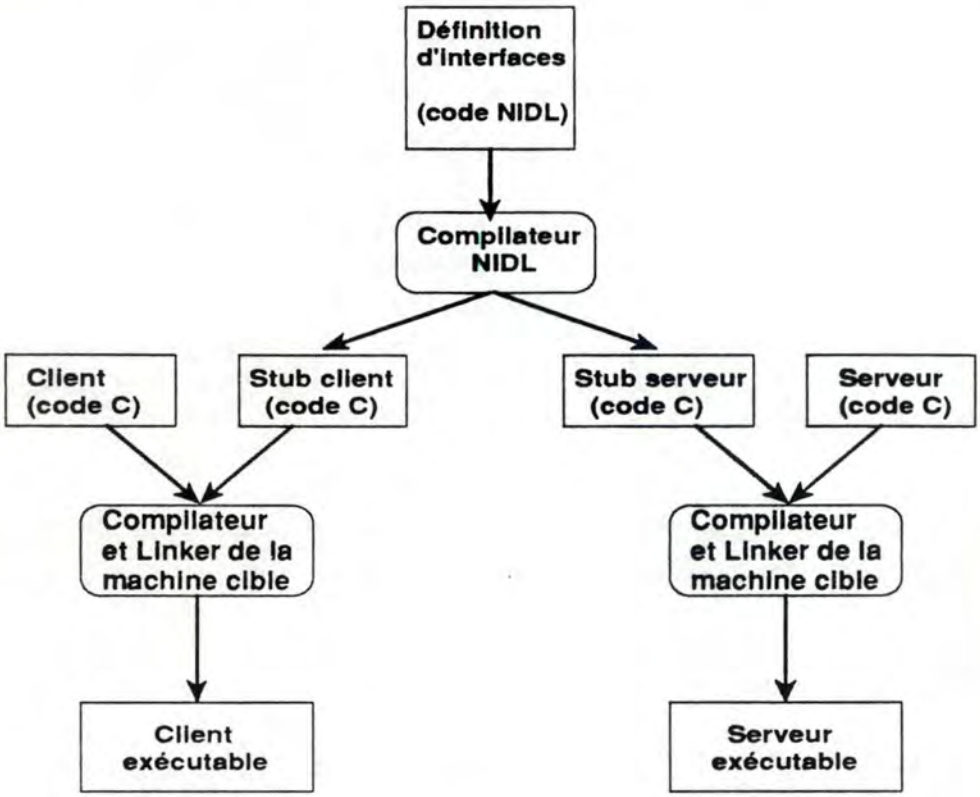

Schéma 9.1 : Développement d'applications sous NCS

Les deux points suivants expliqueront la mise au point des clients et des serveurs d'une application typique. En plus de ceci les annexes disposent , en illustration, d'un exemple simple et d'une liste exhaustive des appels système disponibles. (A. et E.)

9.1 Ecriture du client

9.1.1 Les composantes du client

Le code source contient les éléments suivants :

Un header généré des définitions des interfaces NIDL.
L'application client proprement dite. Elle implémente le client et effectue les appels RPC.
Les fichiers "switch" et "stub" créés par le compilateur NIDL.
Les modules de gestion de binding automatique et de conversion de données intransmissibles.

91

Si l'application client utilise plusieurs interfaces elle est censée inclure, pour chacune d'elles, ses propres fichiers header, switch, stub et ses modules de binding automatique et de conversion.

9.1.2 Squelette de l'application client

L'application client typique se déroule comme exposé ci-dessous.

1) Inclusion des headers propres à l'application et de ceux crées par le compilateur NIDL.

2) Initialisation du Process Fault Manager. Ceci normalise les signaux réceptibles par le client entre machine UNIX/non-UNIX et permet l'installation de *cleanup-handlers*.

3) Localisation du/des serveur/s.

Informations disponibles	Démarche d'identification du serveur
Adresse socket du serveur (saddr)	---------- aucune ----------
Nom du host sur lequel le serveur tourne	- détermination du forwarding port de son LLB - appel de socket_$from_name(...)
Identifiant d'interface, d'objet ou de type d'objet	- consultation auprès du Location Broker - appel de lb_$lookup_interface(...) lb_$lookup_object(...) lb_$lookup_type(...) lb_$lookup_range(...)

4) Binding avec le serveur. Le schéma 9.2 illustre les états de binding des *handles* ainsi que les opérations y liés.

5) Gestion d'erreurs. La table ci-dessous illustre les types d'erreur, sa détection et les actions à entreprendre.

92

Type d'erreur	Détection (contenu de la variable comm_status)	Actions
Communication	rpc_$comm_failure	attendre et réessayer ou arrêter client
Crash du serveur	rpc_$wrong_boot_time rpc_$comm_failure	ou casser ancien binding et refaire nouveau binding
Erreurs d'interface	rpc_$unk_if (interface inconnue) rpc_$op_rng_Error (opération hors portée)	localiser serveur convenable et refaire nouveau binding

L'utilisation de *cleanup handlers* simplifie la gestion d'erreurs. Le module *RPC_RunTime* signale les erreurs au *Process Fault Manager*, qui s'occupe de lancer les handler convenables. Typiquement tout appel RPC est précédé par l'installation d'un de ces handlers. Suite au retour du RPC ce handler sera enlevé.

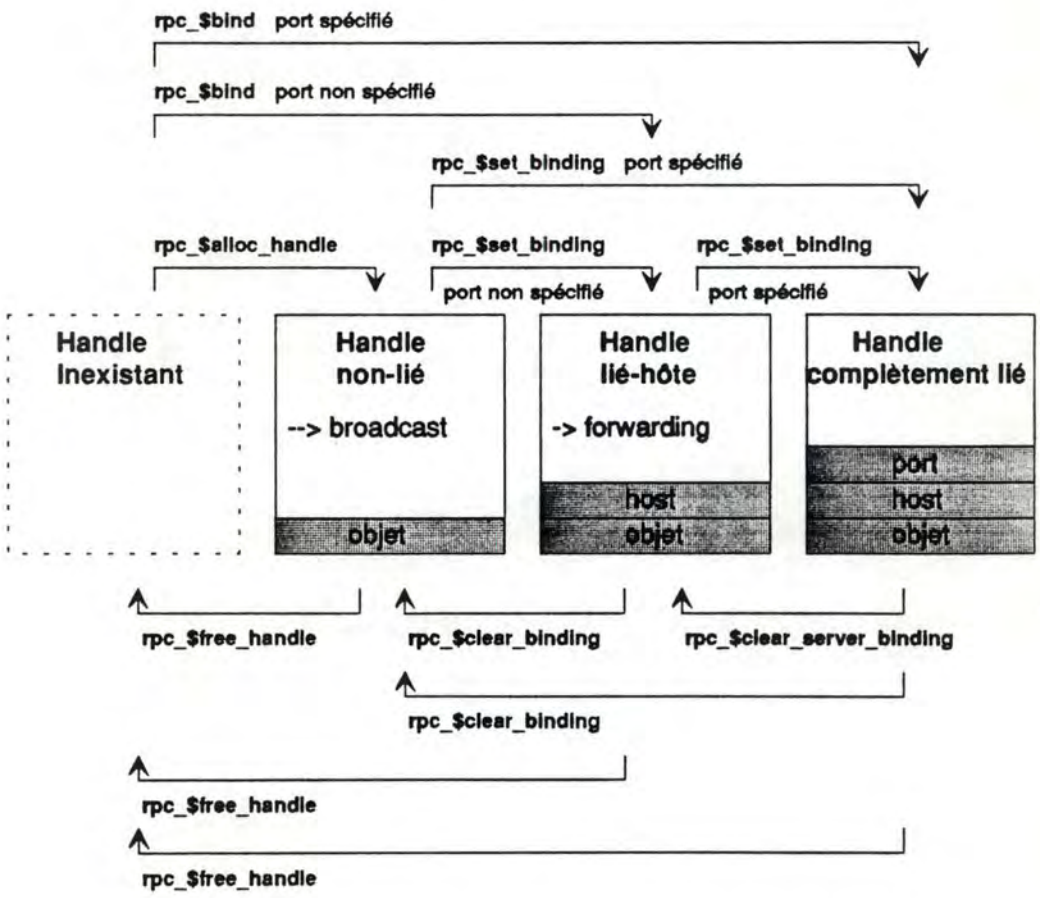

Schéma 9.2 : Gestion des handles et états de binding

93

9.2 Ecriture du serveur

9.2.1 Les composantes du serveur

Un header généré à partir des définitions d'interface par le compilateur NIDL.
Le programme d'initialisation du serveur. Il s'occupe d'enregistrer les interfaces à exporter auprès du RPC_RunTime et du Location Broker.
Le code manager qui implémente les opérations de l'interface remote.
Le stub serveur généré à partir des définitions d'interface NIDL.
Le module de conversions de données complexes en données transmissibles par NCS.

Toute interface exportée par un serveur nécessite l'inclusion de ses propres fichiers header, stub, manager ainsi que des routines de conversion de données.

9.2.2 Ecriture du programme d'initialisation du serveur

1) **Traitement des arguments** fournis au serveur. Dans la ligne de commande on spécifie souvent le type de software réseau sous-jacent et par la aussi la famille d'adressage utilisée. La librairie socket contient des fonctions de conversion d'un nom textuel vers un identificateur de protocole et de validation du protocole sur le host concerné.

2) **Création de sockets** auxquels le serveur "prêtera son oreille". La table liste les fonctions d'ouverture de socket et leur sémantique.

Nom de la fonction	Sémantique
rpc_$use_family	assigner dynamiquement un port opaque au serveur
rpc_$use_family_wk	ouvrir statiquement un port bien connu au serveur

3) **Enregistrement auprès du RPC_RunTime** de tous les objets et managers du serveur. Ceci permet au RunTime de déterminer lors d'un RPC le manager en question et de lui dispatcher l'appel. On y trouve les fonctions suivantes :

Nom de la fonction	Sémantique
rpc_$register_mgr	Enregistrer module manager auprès du RPC_RunTime
rpc_$register_object	Enregistrer nouvel objet auprès du RPC_RunTime

94

4) Enregistrement auprès du location broker. Toute combinaison possible d'objet, d'interface et d'adresse socket nécessite un appel d'enregistrement **lb_$register**.

5) Gestion de fautes et de terminaison du serveur. Lors de la terminaison il s'agit de retirer les enregistrements fait en 3) (**rpc_$unregister**) et 4) (**lb_$unregister**). Ces opérations se font le plus souvent par le mécanisme de *cleanup-handlers*.

6) La mise en état d'écoute du serveur se réalise par l'appel **rpc_$listen** au RunTime.

9.2.3 Ecriture du code manager.

Le manager implémente les fonctions de l'interface pour les objets d'un certain type. En dehors de cela le manager se charge d'une autre série de tâches comme :

1) La définition des EPV , *Entry Point Vectors*. Un *EPV manager* donne des noms aux fonctions implémentées.

2) L'identification d'objets. Le support de plusieurs objets par un manager nécessite l'appel système **rpc_$inq_object** pour identifier l'objet sur lequel le RPC courant s'applique.

3) L'identification des clients peut être intéressante pour l'établissement de fichiers de diagnostic ou de log.

4) Enregistrement d'objets s'avère utile pour un manager gérant ses propres objets transitoires.

5) Initialisation des paramètres statut. Pour toute opération donnant en résultat un statut il faut initialiser la variable **comm_status**.

10. Conclusions sur NCS

10.1 Evaluation du NCS

10.1.1 Network Interface Definition Language

NIDL est un **langage de définition d'interfaces puissant**. Le langage déclaratif, existant en versions PASCAL et C, dispose d'un ensemble plutôt complet de types de données scalaires et construits. La possibilité de conversion de données par l'utilisateur est un atout indispensable pour la distribution d'applications centralisées existantes.

95

Toutefois l'on peut regretter son **support limité des pointeurs**. Le chaînage de pointeurs, leur utilisation dans des structures ou d'autres types composés ainsi que le fait de pointer vers des zones vide (NULL) sont prohibés. Par conséquent le mélange des notations par indices et par pointeurs dans le cadre de tableaux, phénomène bien connu chez les programmeurs en C, ne pourra plus être pratiqué.

Hewlett Packard s'est rendu compte de ces problèmes et promet de remédier à cette situation dans les versions ultérieures à 2.0 [NCS_2]. Les documents draft du NIDL 2.0 [NIDL_2] prévoient le support de nouveaux types de pointeurs qui remédient à la plupart des restrictions enoncées ici. Pour minimiser la taille des paquets l'on a aussi mis au point le mécanisme des *alias*. Au lieu de transférer plusieurs fois les mêmes références de plusieurs pointeurs, on n'en gardera qu'un exemplaire et l'on considère les autres pointeurs comme des *alias*, copies, du premier.

Le compilateur NIDL lui-même est un instrument puissant et extensible. Son extensibilité s'étend à deux niveaux : le support de nouveaux types de données et celui de nouvelles couches de transport. NIDL 2.0 a introduit le type *pipe* pour le transfert efficace de grandes quantités de données structurées.

10.1.2 Network Data Representation

La **philosophie *"receiver makes it right"*** se présente comme une variante intéressante par rapport à l'approche *canonique* préconisée par SUN ou DEC. Elle possède notamment l'avantage d'être **efficace** et de demander un **minimum de conversions** dans un environnement largement hétérogène. Néanmoins elle présente deux désavantages importants. Tout d'abord **certaines architectures ne peuvent être représentées** par le fameux *format label*. Il en est ainsi pour les machines qui ont plus ou moins de 8 bits par byte ou qui connaissent des formats floating-point non IBM,Cray,VAX ou IEEE. Ensuite il faut savoir que **le nombre de conversions à supporter par serveur croit exponentiellement avec le nombre de formats acceptés**. On peut imaginer la complexité des routines de *un/marshalling* qui gonflent considérablement la taille des stubs générés par le compilateur d'interfaces. (voir annexe F.3)

L'*alignement naturel* des données n'est pas le moindre des atouts du NDR. Il s'agit d'un mécanisme efficace favorisant **l'établissement d'interfaces claires dans un environnement hétérogène**.

10.1.3 RPC_RunTime et le modèle réseau

L'objectif primordial de NCS est l'indépendance vis-à-vis des software de communication existants. Le système RunTime constitue une **suite de protocoles hautement fiables** basés sur le concept de *sockets*. (voir annexe F.1). L'usage d'une librairie standardisée de fonctions de gestion et de manipulation des sockets, leur confie un **caractère général et abstrait**. RPC RunTime gère son propre pool de sockets et s'occupe du transport de messages au niveau des applications client et serveur. A ce jour l'on garantit le support des protocoles transport **UDP/IP** et **DOMAIN**.

96

Des raisons de **portabilité et d'efficacité** ont mené les créateurs de NCS à se concentrer sur le support de protocoles non-fiables de transmission par paquets, comme **UDP/IP** et **DOMAIN** [Levy]. On introduisant une couche intermédiaire dans le protocole RPC, NCS évite la barrière de 8K imposée aux paquets par ces couches. Celle-ci s'occupe de trancher les flux de transmission en paquets de taille inférieure à 1K et garde par là, au prix d'une diminution de performance (voir annexe F.2), une certaine **universalité et adaptabilité aux autres protocoles datagramme**.

Or dans certaines situations **un protocole orienté connexion serait désirable**, notamment pour le transport de grandes quantités de données. Si le nombre maximum de clients d'un serveur est connu à priori en reste dans certaines limites, rien ne s'oppose à l'usage de protocoles fiables orientés connexion. Des études (voir annexe F.2) ont montré une **performance supérieure des RPC de SUN_TCP et de DEC** pour une certaine taille des arguments. Les futures versions NCS ne vont pas tarder à fournir le support des *circuits virtuels* de **TCP/IP** et d'**OSI_4**.

10.1.4 Le modèle de binding

Le concept de **binding dynamique** du *location broker*, divisé en domaines locaux et globaux, et la possibilité de **réplication** rencontrent l'attention des développeurs. L'usage **complexe des UUID** se heurte cependant à certains rejets. Le location broker n'accepte que ce type de bas niveau comme clé de recherche. Un service de mapping de noms textuels vers ces identificateurs "hostiles" serait plus qu'approprié.

10.1.5 Sécurité

NCS est en **manque d'un service de sécurité fiable et efficace**. Celui-ci est censé se situer, dans le modèle NCA, au niveau des *network broker*. L'absence de sécurisation risque de freiner l'acceptation du NCS dans certains domaines sensibles. Les implémentations à venir ont prévu les support de plusieurs mécanismes d'authentification RPC, parmi lesquels se trouve le système **KERBEROS** (1).

10.2 L'avenir du NCS

NCS propose une implémentation **générale, fiable et efficace** d'un modèle de système distribué. NCS est actuellement disponible en version 1.5 et de nombreux constructeurs sont en phase de portage de la version 2.0 perfectionnée et corrigée en de nombreux points. NCS propose une démarche cohérente ainsi qu'un *package* complet d'outils facilitant le développement de nouvelles applications distribuées ainsi que la conversions d'anciennes applications centralisées. NCS est un élément central du Distributed Computing Environment d'OSF/1, système d'exploitation qui risque, à moyen terme, de prendre une place centrale dans le monde de l'informatique.

(1) Kerberos est un système d'authentification mis au point par le MIT dans le cadre du projet Athena. Sa philosophie se base sur l'existence d'un troisième agent, distributeur de clés d'authentification, lors des interactions client-serveur. (voir 3.4.2 b. du chap. I)

97

Conclusion :

Le *Remote Procedure Call* est un outil intéressant pour le développement, la mise en place et l'exécution d'applications dans les systèmes distribués. Il propose une alternative élégante à la diffusion incontrôlée de protocoles application, en restant fidèle à la démarche procédurale simple. Néanmoins, comme tout outil, le RPC est sujet à certaines limitations.

Tout d'abord il y a le problème de *transparence*. Le développeur devra adapter quelque peu son style de programmation en essayant de découpler entièrement les environnements des applications principales et des sous-routines. Les références à un environnement global, les pointeurs posent des problèmes graves, cela surtout lors des tentatives de distribution d'applications centralisés existantes. Il est clair que toute avance au niveau de la transparence va à l'encontre des objectifs de performance et d'efficacité. Va-t-on suivre les références et transférer les objets référencés dans l'appel RPC, transférer une fois pour toutes tout l'environnement global ou bien signaler une erreur et court-circuiter tous les efforts de transparence ? Voilà des choix à prendre par les concepteurs de l'outil RPC.

Qu'en est-il des types complexes non transmissibles par les stubs ? Un compilateur d'interfaces ne pourra jamais prévoir tous les cas de figure possibles, sauf au prix d'un "gonflage" excessif de la complexité du compilateur et de la taille des stubs. Souvent les concepteurs ont prévu la possibilité d'écriture de routines de conversion vers des types transmissibles. La linéarisation des arguments par les stubs nécessite la connaissance exacte de leur type. Or le langage "C", par exemple, dispose du type *union* dont le type de données n'est pas défini univoquement. La génération automatique d'un stub pour la fonction standard *printf* constitue une épreuve rude pour tout compilateur d'interfaces existant. De nouvelles entraves à la transparence !

La *performance* est un autre facteur à considérer. L'approche RPC requiert la suspension du client jusqu'à l'arrivée de la réponse. En fonction de l'architecture de communication utilisée et de la charge du système le RPC risque de devenir très coûteux et les avantages liés à l'exécution lointaine sur des processeurs spécialisés peuvent s'annuler. Un parallélisme plus poussé nécessite des efforts supplémentaires du programmeur.

La *gestion des erreurs* et la garantie d'une sémantique identique aux appels locaux imposent une charge lourde au protocole RPC. Le protocole ne pourra cacher tous les types d'erreur à l'application. La procédure appellante peut, par exemple, disposer d'informations d'état à modifier en cas d'erreur.

Cette pluie de critiques n'est pas destinée au découragement des développeurs. Le RPC est un instrument fantastique mais loin d'une solution universelle et parfaite. Le développeur devrait toutefois être conscient de limites de l'outil et prévoir des efforts intellectuels supplémentaires.

Bibliographie

[ANSA] Alvey Advanced Network Systems Architecture Project
"ANSA Reference Manual 0.03 (Draft)"
1987 Alvey Cambridge

[AP] Array Processors

1. ICL DAP :
Hockney & Jessop
"Parallel Computers"
1986 Adam Hilger (Bristol)

2. Illiac4 :
Bouknight S. et al.
"The ILLIAC IV system"
1981

3. Connection Machine :
Hillis D.
"The Connection Machine"
1985 MIT Press

[BiNe] Andrew D. Birrell & Bruce Jay Nelson
"Implementing Remote Procedure Calls"
02/1984, ACM Transactions on Computer Systems

[ChDu] Chambers, Duce et al.
"Distributed Computing"
1984 London Academic Press

Duce
"Distributed Computing Systems Programme"
1984 London : Peter Peregrinus

[CoDo] George F. Colouris & Jean Dollimore
"DISTRIBUTED SYSTEMS, Concepts and Design"
1989 Addison Wesley, ISBN 0-201-18059-6

[CPS] Apollo Computer Inc.
"Concurrent Programming Support Reference"
Apollo Inc. , order number : 010233

[DCE_ADG] Open Software Foundation
"DCE Application Development Guide"
15/09/1990, OSF DCE Release 1.0 S2

[DCE_PW] BULL S.A. France, System Emgineering
"PURPLE WAY"
12/1990, BULL S.A. Massy / Paris

[Denning] Denning D.
"Cryptography and Data Security"
1982 Addison Wesley

[GrDe] Graham G.S. & Denning P.J.
"Protection - principles and practice"
1972 AFIPS Spring Joint Computer Conference

[Grosch] Grosch H.R.J.
"High speed arithmetic : the digital computer as a research tool"
1953 Optical Society of America, vol. 4 no. 4

[Harel] Dov Harel
"A Comparative Study of Some RPC Compilers"
01/1989

[Hoare] Hoare C.A.R.
"Monitors: an operating system structuring concept"
1974 Communications of the ACM vol. 17 no. 10

[Kahn] Kahn D.
"The Codebreakers"
1967 AFIPS National Computer Conference

[Lampson] Lampson B.W.
"Dynamic protection structures"
1969 AFIPS Fall Joint Computer Conference

[LaSp] Lampson & Sproull
"An open operating system for a single-user machine"
1979, 7th symposium on Operating Systems Principles

[LCS] Loosely Coupled Systems

1. LOCUS :
Popek, Walker et al.
"The LOCUS Distributed System Architecture"
1985, MIT Press

2. Newcastle Connection :
Brownbridge et al.
"The Newcastle Connection, or UNIXes
of the world unite !"
Software - Practice and Experience vol. 12

3. Amoeba :
Mullender S.J.
"Principles of Distributed
Operating System Design" (doct. thesis)
1985, Mathematisch Centrum Amsterdam

Mullender S.J. and Tanenbaum A.S.
"The design of a capability based
distributed operating system"
1986 Computer Journal vol. 29 no. 4

4. Cambridge Distributed Computing System :
Needham & Herbert
"The Cambridge Distributed Computing System"
Addison Wesley

5. VAX Cluster :
Kronenberg et al.
"VAX clusters, a closely-coupled
distributed system"
1981 ACM Operating Systems Review
vol. 19 no. 5

[LeLann] LeLann G.
"Motivations, objectives and characterization of distributed systems"
1981 Lampson : Distributed Systems - Architecture and Implementation
Springer Verlag

[Levy] Joshua Levy
"A Comparison of Commercial RPC Systems"
1989, Atherton Technology

[NBS_DES] National Bureau of Standards
"Data Encryption Standard"
1977 FIP no. 46

[NCA] Lisa Zahn et al.
"Network Computing Architecture"
1990 Prentice Hall, ISBN 0-13-611674-4

[NCA_2] Hewlett-Packard Co. / Digital Equipment Co.
"Network Computing Architecture Version 2.0 Specifications"
10/07/1990, Preliminary update to [NCA]

[NCS] Mike Kong et al.
"Network Computing System - Reference Manual"
1990 Prentice Hall, ISBN 0-13-617085-4

[NCS_TUT] Nathaniel Mishkin & Paul J. Leach
"Network Computing System Tutorial"
1988-1990, Apollo Computer Inc.

[NeSch] Needham R.M. & Schroeder M.D.
"Using encryption for authentication in large networks of computers"
1978 Communication of the ACM vol. 21

[News] SUN Microsystems Inc.
"Sun Network-extensible Window System (NeWS)"
1987 Sun Microsystems Inc.

[NIDL_2] Hewlett-Packard Co. / Digital Equipment Co.
"Functional Specification of the NIDL language"
16/07/1990, Draft Document

[OPA] Other Parallel Architectures

1. dataflow architecture :
Dennis J.B.
"1st version of a dataflow procedure language"
1974 Lectures in Computer Science no. 19 , Springer Verlag

Treleaven et al.
"Data driven and demand driven computer architecture"
1982 Computer Surveys vol. 14 no. 1

2. functional architectures :
Darlington J. & Reeves M.J.
"Alice: a multiprocessor reduction system for applicative languages"
1981 ACM/MIT Conf. on Functional Languages and Computer Architecture

[RiShAd] Rivest R.L., Shamir A. & Adelman
"A method of obtaining digital signatures and public key cryptosystems"
1978 Communications of the ACM vol. 21 no. 2

[Sansom] Sansom R.D., Julin D.P. & Rashid R.F.
"Extending a capability based system into a network environnment"
1986 Technical Report Carnegie-Mellon Univ.

[SchGet] Scheifler R.W. & Gettys J.
"The X window system"
1986 ACM Transactions on Graphics vol. 5 no. 2

[ShHu] Shoch J.F. & Hupp J.A.
"The 'Worm' programs - early experience with a distributed computation"
1982 Communications of the ACM vol. 25 no. 3

[SUN] Sun Microsystems Inc.
"Open Network Computing"
1988 Sun Microsystems Inc.

[Tanenbaum] Tanenbaum A.S.
"Computer Networks"
1981 Prentice Hall

[TCS] Tightly Coupled Systems

1. C.MMP :
Wulf et al.
"C.mmp : a multi-processor"
1972 AFIPS Fall Joint Computer Conference vol. 41 no. 2

2. CM* :
Jones et al.
"Software managment of CM*: a multiple processor"
1977 AFIPS National Computer Conference

3. P-N machine :
Iliffe J.K.
"Advanced Computer Design"
1982 Prentice-Hall

[Unix] Ritchie D.M. & Thompson K.
"The UNIX time-sharing system"
1974 Communications of the ACM vol. 17 no. 7

Jean-Marie Rifflet
"La programmation sous UNIX", 2nde édition
1989 Mc Graw Hill, ISBN 2-7042-1184-4

Jean-Marie Rifflet
"La communication sous UNIX"
1990 Mc Graw Hill, ISBN 2-7642-1240-6

[XEROX] Xerox Corporation
"Courier : the remote procedure call protocol"
1981 Xerox Corporation

Annexe A :

Les interfaces NCS

Annexe A.1 :

Conversation Manager

Chapter 8

Conversation Manager Interface

The rule for non-idempotent operations is that all non-idempotent operations are executed by the server "at most once"; that is, not at all or exactly once. The protocol that the NCA/RPC facility uses to enforce this rule is called the **callback mechanism**. The server implementation of the callback mechanism occurs in the server FSM: the protocol for non-idempotent requests requires the server to perform a callback when it receives a request from a client about which it has no information. The server makes the callback request by making a remote procedure call to the Conversation Manager at the client side; see Chapter 7 for the definition of the server callback protocol.

The Conversation Manager is the client's implementation of the callback mechanism. In addition to implementing the three FSMs defined in Chapter 6, an NCA/RPC client is required to listen for server callback requests and to implement and export the Conversation Manager interface to process them.

The Conversation Manager is an NCA-defined remote interface that processes server callback requests and returns information to the server that it uses to validate the request it has received against the client's record of its current outstanding request. The Conversation Manager runs in the calling client activity and sends the results of the callback to the socket from which the client's original request was made. The Conversation Manager is defined in an interface definition named **conv.idl**; its contents are as follows.

```
%pascal

[uuid(333a22760000.0d.00.00.80.9c.00.00.00), version(3)]

interface conv_;
import
    'nbase.idl';

[idempotent] procedure conv_$who_are_you(
                            in      h:          handle_t;
                            in ref  actuid:     uuid_$t;
                            in      boot_time:  unsigned32;
                            out     seq:        unsigned32;
                            out     st:         status_$t
                            );
end;
```

The Conversation Manager interface definition

1. Imports the interface definition file **nbase.idl**, which defines data types that the Conversation Manager requires; Appendix B gives the complete contents of **nbase.idl**.

2. Defines one idempotent operation named **conv_$who_are_you**. The next pages give syntax and usage information for **conv_$who_are_you**.

NAME

conv_$who_are_you — Implements the callback mechanism for the NCA/RPC client.

SYNOPSIS (NIDL/Pascal)

```
[idempotent] procedure conv_$who_are_you(
                    in      handle:     handle_t;
                    in ref  actuid:     uuid_$t;
                    in      boot_time:  unsigned32;
                    out     seqnum:     unsigned32;
                    out     status:     status_$t
                    );
```

DESCRIPTION

The **conv_$who_are_you** operation is an idempotent procedure that takes a calling client activity UUID and the server's record of its boot time as input, and returns the current sequence number held by the calling client and status information as output to the server making the callback.

handle — A primitive handle. See Chapter 9 for a description of primitive handles and the **handle_t** type.

actuid — The activity UUID of the calling client; in implementations that support multiple simultaneous client requests, this value is used to identify the client about whose request the server is querying.

boot_time — A 32-bit integer that identifies the time at which the server last booted. The operation stores this value in the client FSM global variable bootTime.

seqnum — A 32-bit integer that identifies sequence number associated with the client's current outstanding request.

status — Status information returned by the operation to the server that called it, in **status_$t** format; the **status_$t** type is defined in **nbase.idl**. Possible values are

YouCrashed — The server has crashed and rebooted since establishing communications with the client. The hexadecimal value for this error is defined in Table 4–6 and in the interface definition file **ncastat.idl** (Appendix C).

NotInCall — The client about whom the server is querying does not have an outstanding request in progress.

Normally, a server can detect request duplication by comparing the incoming sequence number against its record of a client's previous sequence number. However, there are three cases in which a server will have no record of a client sequence number:

- When the request is the first request from a client
- When the server has executed the request but (due to delay in client acknowledgement) has discarded all information about the client
- When the server has executed the request, but has crashed before sending the response and thus has lost all information about the client

The client and server use the *boot_time* and *seqnum* values passed between them via **conv_$who_are_you** to detect duplicate requests for non-idempotent operations in the face of acknowledgment delays or server crashes.

Acknowledgment Delays

As described in the NCA protocol summary given in Chapter 1 (and defined in the FSM tables in Chapter 7) if the server has not heard from a client for a lengthy period of time (or has run out of storage space), it discards all information about the client (that is, the response and the sequence number for that response). Consequently, when it receives a request from a client for which it has no sequence number, the server cannot determine whether the request is the first from this client, or whether it has heard from the client a long time ago and since discarded any information it formerly had about it. From the server's point of view, the received request could be a duplicate packet which it has already processed.

The server handles this case by making a remote call to the **conv_$who_are_you** operation at the calling client, passing its current boot time in *boot_time*. When it receives the callback, **conv_$who_are_you** stores the server's boot time on the client's behalf in the client FSM global variable bootTime (see Table 6-6) and sends back the client's current sequence number in *seqnum* to the server. The server compares the value of the sequence number returned in the callback to the sequence number in the client's original request. If they are identical, the server executes the original request; if not, it ignores the request. In this case, it is the server that enforces the "at most once" rule by means of the sequence number passed in the callback.

Server Crashes

A rebooted server has no sequence number about a client because it has lost all information about it as a result of the crash. Consequently, if a request arrives, the rebooted server is unable to determine whether the request is new, or whether it executed the request just before it crashed. The callback mechanism detects request duplication in this situation as follows.

1. The rebooted server must call back the client upon receipt of the request, since it will not have a sequence number.
2. The client possesses the crashed server's boot time recorded during the callback. (Since the case assumes that the crashed server has executed the call, the server must have previously called back the client.)
3. The server sends its rebooted server boot time in *boot_time* to the **conv_$who_are_you** operation. By comparing its record of the server's boot time with the value it receives in *boot_time*, **conv_$who_are_you** (or the client) can determine whether a new version of the server is calling it back. If this is so, it sends the YouCrashed error in its response to the callback. When the server receives this message, it rejects the request.

In the case of server crashes, it is the client that enforces the "at most once" rule by means of the server boot time passed in the callback.

Annexe A.2 :

Data Replication Manager (DRM)

glb_drm.idl

```
{ glb_drm - global location broker DRM interface }

[uuid(339a6e4fe000.0d.00.00.87.84.00.00.00), version(1)]
interface glb_drm_;

import
        'glb.idl',
        'drm_base.idl';

type
        lb_$entry_key_t = record
                object          : uuid_$t;
                obj_type        : uuid_$t;
                obj_interface   : uuid_$t;
                saddr_len       : integer32;
                saddr           : socket_$addr_t;
                end;
        lb_$drm_entry_t = record
                hdr             : drm_$entry_hdr_t;
                data            : lb_$entry_t;
                end;
        lb_$drm_entries_t = array[1..*] of lb_$drm_entry_t;

procedure glb_drm_$svr_add (
        in      h       : handle_t;
        in ref  object  : uuid_$t;
        in ref  entry   : lb_$entry_t;
        in ref  origin  : drm_$origin_t;
        in ref  time    : time_$clock_t;
        out     time2   : time_$clock_t;
        out     st      : status_$t
    );

procedure glb_drm_$svr_del (
        in      h       : handle_t;
        in ref  object  : uuid_$t;
        in ref  key     : lb_$entry_key_t;
        in ref  origin  : drm_$origin_t;
        in ref  time    : time_$clock_t;
        out     time2   : time_$clock_t;
        out     st      : status_$t
    );
```

glb_drm.idl, cont'd.

```
const glb_drm_$max_read_results = 5;

procedure glb_drm_$svr_read (
        in      h       : handle_t;
        in ref  object  : uuid_$t;
        in out  start   : drm_$entry_ptr_t;
        in      max_ents: linteger;
        out     nents   : linteger;
        out     entries : [ max_is(max_ents), last_is(nents) ]
                          lb_$drm_entries_t;
        out     st      : status_$t
    );
end;
```

rdrm.idl

```
[uuid(33599c670000.0d.00.00.24.34.00.00.00), version(1)]
interface rdrm_;

import
    'socket.idl',
    'rpc.idl',
    'drm_base.idl';

type
   drm_$merge_type_ptr_t = ^drm_$merge_type_t;

procedure rdrm_$info(
        in h: handle_t;
        in ref object: uuid_$t;
        out info: drm_$info_t;
        out st: status_$t
    );

procedure rdrm_$reset_replica(
        h: handle_t;
        in ref object: uuid_$t;
        out st: status_$t
    );

procedure rdrm_$close_replica(
        h: handle_t;
        in ref object: uuid_$t;
        out st: status_$t
    );

procedure rdrm_$add_replica(
        h: handle_t;
        in ref object: uuid_$t;
        in ref rep_addr: socket_$addr_t;
        in rep_addr_len: integer32;
        out st: status_$t
    );

procedure rdrm_$rep_replica(
        h: handle_t;
        in ref object: uuid_$t;
        in ref rep_addr: socket_$addr_t;
        in rep_addr_len: integer32;
        out st: status_$t
    );
```

rdrm.idl, cont'd.

```
procedure rdrm_$del_replica(
        h: handle_t;
        in ref object: uuid_$t;
        in ref rep_id: socket_$host_id_t;
        in rep_id_len: integer32;
        out st: status_$t
    );

procedure rdrm_$svr_add_replica(
        h: handle_t;
        in ref object: uuid_$t;
        in ref rep_addr: socket_$addr_t;
        in rep_addr_len: integer32;
        in ref rep_origin: drm_$origin_t;
        in ref svr_time: time_$clock_t;
        out local_replist_add_time: time_$clock_t;
        out st: status_$t
    );

procedure rdrm_$svr_del_replica(
        h: handle_t;
        in ref object: uuid_$t;
        in ref rep_id: socket_$host_id_t;
        in rep_id_len: integer32;
        in ref rep_origin: drm_$origin_t;
        in ref svr_time: time_$clock_t;
        out local_replist_add_time: time_$clock_t;
        out st: status_$t
    );

procedure rdrm_$read_replicas(
        h: handle_t;
        in ref object: uuid_$t;
        in out start_ent: drm_$replica_ptr_t;
        in max_ents: linteger;
        out n_ents: linteger;
        out replist: [last_is(n_ents)] drm_$replist_t;
        out st: status_$t
    );
```

rdrm.idl, cont'd.

```
procedure rdrm_$svr_read_replicas(
        h: handle_t;
        in ref object: uuid_$t;
        in out start_ent: drm_$replica_ptr_t;
        in max_ents: linteger;
        out n_ents: linteger;
        out replist: [last_is(n_ents)] drm_$replist_rec_t;
        out st: status_$t
    );

procedure rdrm_$merge(
        h: handle_t;
        in ref object: uuid_$t;
        in merge_type: drm_$merge_type_ptr_t;
        in ref rem_rep_addr: socket_$addr_t;
        in rem_rep_addr_len: integer32;
        out st: status_$t
    );
end;
```

Annexe A.3 :

RPC Error

Chapter 9

error_$ Calls

Contents

error_$intro

NAME

error_$intro – error text database operations

DESCRIPTION

The **error_$** calls convert status codes into textual error messages.

There is no header file for the **error_$** calls. They can be declared as follows:

```
extern void error_$c_get_text();
extern char *error_$c_text();
```

The **error_$** calls use the **status_$t** data type, which is defined in <idl/c/nbase.h>.

DATA TYPES

The **error_$** calls take as input a status code in **status_$t** format.

status_$t A status code. Most of the NCS calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 mode : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

mode This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

error_$c_get_text

NAME

error_$c_get_text – return subsystem, module, and error texts for a status code

SYNOPSIS (C)

```
void error_$c_get_text(
        status_$t status,
        char *subsys,
        long subsysmax,
        char *module,
        long modulemax,
        char *error,
        long errormax)
```

SYNOPSIS (PASCAL)

```
procedure error_$c_get_text(
        in status: status_$t;
        out subsys: univ char;
        in subsysmax: integer32;
        out module: univ char;
        in modulemax: integer32;
        out error: univ char;
        in errormax: integer32);
```

DESCRIPTION

The **error_$c_get_text** call returns predefined text strings that describe the subsystem, the module, and the error represented by a status code. The strings are null terminated.

status A status code in **status_$t** format.

subsys A character string. The subsystem represented by the status code.

subsysmax The maximum number of bytes to be returned in *subsys*.

module A character string. The module represented by the status code.

modulemax The maximum number of bytes to be returned in *module*.

error A character string. The error represented by the status code.

errormax The maximum number of bytes to be returned in *error*.

EXAMPLE

The following statement returns text strings for the subsystem, module, and error represented by the status code st:

```
error_$c_get_text (st, subsys, MAX, module, MAX, error, MAX);
```

SEE ALSO

error_$c_text

error_$c_text

NAME

error_$c_text – return an error message for a status code

SYNOPSIS (C)

```
char *error_$c_text(
        status_$t status,
        char *message,
        int messagemax)
```

SYNOPSIS (PASCAL)

```
procedure error_$c_text(
        in status: status_$t;
        out message: univ char;
        in messagemax: integer32);
```

DESCRIPTION

The **error_$c_text** call returns a null-terminated error message for reporting the completion status of a call. The error message is composed from predefined text strings that describe the subsystem, the module, and the error represented by the status code.

status A status code in **status_$t** format.

message A character string. The error message represented by the status code.

messagemax
The maximum number of bytes to be returned in *message*.

EXAMPLE

The following statement returns an error message for reporting the status code st:

```
error_$c_text (st, message, MAX);
```

SEE ALSO

error_$c_get_text

Chapter 10

lb_$ Calls

Contents

Annexe A.4 :

Location Broker

NAME

lb_$intro – interface to the Location Broker

SYNOPSIS (C)

```
#include <idl/c/lb.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/lb.ins.pas';
```

DESCRIPTION

The **lb_$ calls** constitute the programmatic interface to the Location Broker Client Agent. This interface is defined by the file *idl*/**lb.idl**, where the symbol *idl* denotes the system idl directory. On Apollo workstations and other UNIX systems, *idl* is /usr/include/idl.

EXTERNAL VARIABLES

The following external variable is used in **lb_$** lookup calls:

uuid_$nil
: An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

CONSTANTS

The following constants are used in **lb_$ calls**:

lb_$default_lookup_handle
: A value for **lb_$lookup_handle_t**. On input, it specifies that a lookup is to start searching at the beginning of the database. On output, it indicates that a lookup reached the end of the database.

lb_$mod A module code indicating the Location Broker module. See the description of the **status_$t** type.

lb_$server_flag_local
: Used in the **flags** field of an **lb_$entry_t** variable. Specifies that an entry is to be registered only in the Local Location Broker (LLB) database. See the description of lb_$server_flag_t in the "DATA TYPES" section.

DATA TYPES

This section describes data types used in **lb_$ calls**.

lb_$entry_t
: An identifier for an object, a type, an interface, and the socket address used to access a server exporting the interface to the object. The **lb_$entry_t** type is defined as follows:

```
typedef struct lb_$entry_t lb_$entry_t;
struct lb_$entry_t {
    uuid_$t object;
    uuid_$t obj_type;
    uuid_$t obj_interface;
    lb_$server_flag_t flags;
    ndr_$char annotation[64];
    ndr_$ulong_int saddr_len;
    socket_$addr_t saddr;
};
```

object A **uuid_$t**. The UUID for the object. Can be **uuid_$nil**.

obj_type A **uuid_$t**. The UUID for the type of the object. Can be **uuid_$nil**.

obj_interface
: A **uuid_$t**. The UUID for the interface. Can be **uuid_$nil**.

flags An **lb_$server_flag_t**. Must be 0 or **lb_$server_flag_local**. A value of 0 specifies that the entry is to be registered in both the Local Location Broker (LLB) and Global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

annotation
: A 64-character array. User-defined textual annotation.

saddr_len
: A 32-bit integer. The length of the **saddr** field.

saddr A **socket_$addr_t**. The socket address of the server.

lb_$lookup_handle_t
: A 32-bit integer used to specify the point in the database at which a Location Broker lookup operation will start.

lb_$server_flag_t
: A 32-bit integer used to specify the Location Broker databases in which an entry is to be registered. A value of 0 specifies registration in both the LLB and GLB databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

socket_$addr_t
: A socket address record that uniquely identifies a socket. See the **socket_$intro** section of Chapter 15 for a full description of this type.

status_$t A status code. Most of the NCS calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. See the **uuid_$intro** section of Chapter 16 for a full description of this type.

STATUS CODES

The following are status codes returned by **lb_$** calls:

lb_$cant_access
The Location Broker cannot access the database. Among the possible reasons: (1) the database does not exist, and the Location Broker cannot create it; (2) the database exists, but the Location Broker cannot access it; (3) the GLB entry table or the GLB propagation queue is full.

lb_$database_busy
The Location Broker database is currently in use in an incompatible manner.

lb_$database_invalid
The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$not_registered
The Location Broker does not have any entries that match the criteria specified in the lookup or unregister call. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an **lb_$lookup_object_local** or **lb_$lookup_range** call specifying an LLB, check that you have specified the correct LLB.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

lb_$update_failed
The Location Broker was unable to register or unregister the entry (for example, because the broker ran out of disk space).

status_$ok
The call was successful.

FILES

idl/lb.idl

NAME

lb_$lookup_interface – look up information about an interface in the GLB database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$lookup_interface(
        uuid_$t *obj_interface,
        lb_$lookup_handle_t *lookup_handle,
        unsigned long max_results,
        unsigned long *num_results,
        lb_$entry_t results[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$lookup_interface(
        in obj_interface: uuid_$t;
        in out lookup_handle: lb_$lookup_handle_t;
        in max_results: unsigned32;
        out num_results: unsigned32;
        out results: array [1..*] of lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$lookup_interface** call returns GLB database entries whose **obj_interface** fields match the specified interface. It returns information about all replicas of all objects that can be accessed through that interface.

The **lb_$lookup_interface** call cannot return more than *max_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup calls to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the positions of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup call will skip or duplicate entries.

obj_interface

The UUID of the interface being looked up.

lookup_handle

A position in the database.

On input, the *lookup_handle* indicates the position in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found *max_results* matching entries before it reached the end of the database.

max_results

The maximum number of entries that can be returned by a single call. This should be the number of elements in the *results* array.

num_results

The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specified by the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status The completion status.

EXAMPLE

The following statement looks up information in the GLB database about the directory interface identified by **dirid**:

```
lb_$lookup_interface (&dirid, &lookup_handle, max_results,
    &num_results, results, &st);
```

FILES

idl/lb.idl

SEE ALSO

lb_$lookup_object, lb_$lookup_range, lb_$lookup_type

NAME

lb_$lookup_object – look up information about an object in the GLB database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$lookup_object(
        uuid_$t *object,
        lb_$lookup_handle_t *lookup_handle,
        unsigned long max_results,
        unsigned long *num_results,
        lb_$entry_t results[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$lookup_object(
        in object: uuid_$t;
        in out lookup_handle: lb_$lookup_handle_t;
        in max_results: unsigned32;
        out num_results: unsigned32;
        out results: array [1..*] of lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$lookup_object** call returns GLB database entries whose object fields match the specified type. It returns information about all replicas of an object and about all interfaces to the object.

The **lb_$lookup_object** call cannot return more than *max_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup calls to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the positions of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup call will skip or duplicate entries.

object The UUID of the object being looked up.

lookup_handle

A position in the database.

On input, the *lookup_handle* indicates the position in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found *max_results* matching entries before it reached the end of the database.

max_results

The maximum number of entries that can be returned by a single call. This shou number of elements in the *results* array.

num_results

The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specifi *max_results* parameter. If the array contains any entries for servers on the local netwo entries appear first.

status The completion status.

EXAMPLE

The following statement looks up GLB database entries for the object identified by **objid**:

```
lb_$lookup_object (&objid, &lookup_handle, max_results,
    &num_results, results, &st);
```

FILES

idl/lb.idl

SEE ALSO

lb_$lookup_interface, lb_$lookup_object_local, lb_$lookup_range, lb_$lookup_type

NAME

lb_$lookup_object_local – look up information about an object in an LLB database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$lookup_object_local(
        uuid_$t *object,
        socket_$addr_t *location,
        unsigned long location_length,
        lb_$lookup_handle_t *lookup_handle,
        unsigned long max_results,
        unsigned long *num_results,
        lb_$entry_t results[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$lookup_object_local(
        in object: uuid_$t;
        in location: socket_$addr_t;
        in location_length: unsigned32;
        in out lookup_handle: lb_$lookup_handle_t;
        in max_results: unsigned32;
        out num_results: unsigned32;
        out results: array [1..*] of lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$lookup_object_local** call searches the specified LLB database and returns all entries whose object fields match the specified object. It returns information about all replicas of an object and all interfaces to the object that are located on the specified host.

The **lb_$lookup_object_local** call cannot return more than *max_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup calls to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the positions of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup call will skip or duplicate entries.

object The UUID of the object being looked up.

location The location of the LLB database to be searched. The socket address must specify the network address of a host. However, the port number in the socket address is ignored, and the lookup request is always sent to the LLB port.

location_length
The length, in bytes, of the socket address specified by the location field.

lookup_handle
A position in the database.

On input, the *lookup_handle* indicates the position in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found *max_results* matching entries before it reached the end of the database.

max_results
The maximum number of entries that can be returned by a single call. This should be the number of elements in the *results* array.

num_results
The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specified by the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status The completion status.

EXAMPLE

The following statement looks up the replicated object identified by **repobjid** in the LLB database at the host specified by **loc**. Though this object is replicated, only one replica is located on any host, so the call will return at most one result.

```
lb_$lookup_object_local (&repobjid, &loc, loc_len, &lookup_handle,
    1, &num_results, results, &st);
```

FILES

*idl/***lb.idl**

SEE ALSO

lb_$lookup_range

lb_$lookup_range

NAME

lb_$lookup_range – look up information in a GLB or LLB database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$lookup_range(
        uuid_$t *object,
        uuid_$t *obj_type,
        uuid_$t *obj_interface,
        socket_$addr_t *location,
        unsigned long location_length,
        lb_$lookup_handle_t *lookup_handle,
        unsigned long max_results,
        unsigned long *num_results,
        lb_$entry_t results[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$lookup_range(
        in object: uuid_$t;
        in obj_type: uuid_$t;
        in obj_interface: uuid_$t;
        in location: socket_$addr_t;
        in location_length: unsigned32;
        in out lookup_handle: lb_$lookup_handle_t;
        in max_results: unsigned32;
        out num_results: unsigned32;
        out results: array [1..*] of lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$lookup_range** call returns database entries whose **object**, **obj_type**, and **obj_interface** fields match the specified values. A value of **uuid_$nil** in any of these input parameters acts as a wildcard and will match any value in the corresponding entry field. You can specify wildcards in any combination of these parameters.

The **lb_$lookup_range** call cannot return more than *max_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup calls to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the positions of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup call will skip or duplicate entries.

lb_$lookup_range

object The UUID of the object being looked up.

obj_type The UUID of the type being looked up.

obj_interface
The UUID of the interface being looked up.

location The location of the database to be searched. If the value of *location_length* is 0, the GLB database is searched. Otherwise, the LLB database at the host specified by the socket address is searched; in this case, the port number in the socket address is ignored, and the lookup request is sent to the LLB port.

location_length
The length, in bytes, of the socket address specified by the *location* field. A value of 0 indicates that the GLB database is to be searched.

lookup_handle
A position in the database.

On input, the *lookup_handle* indicates the position in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found *max_results* matching entries before it reached the end of the database.

max_results
The maximum number of entries that can be returned by a single call. This should be the number of elements in the *results* array.

num_results
The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specified by the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status The completion status.

EXAMPLE

The following statement looks up information in the GLB database about servers that export the interface identified by **matrix_id** for any objects of the type identified by **array_id**. The variable **glb** is defined elsewhere as a null pointer.

```
lb_$lookup_range (&uuid_$nil, &array_id, &matrix_id, glb, 0,
    &lookup_handle, max_results, &num_results, results, &st);
```

FILES

idl/lb.idl

SEE ALSO

lb_$lookup_interface, lb_$lookup_object, lb_$lookup_object_local, lb_$lookup_type

lb_$lookup_type

NAME

lb_$lookup_type – look up information about a type in the GLB database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$lookup_type(
        uuid_$t *obj_type,
        lb_$lookup_handle_t *lookup_handle,
        unsigned long max_results,
        unsigned long *num_results,
        lb_$entry_t results[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$lookup_type(
        in obj_type: uuid_$t;
        in out lookup_handle: lb_$lookup_handle_t;
        in max_results: unsigned32;
        out num_results: unsigned32;
        out results: array [1..*] of lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$lookup_type** call returns GLB database entries whose **obj_type** fields match the specified type. It returns information about all replicas of all objects of that type and about all interfaces to each of these objects.

The **lb_$lookup_type** call cannot return more than *max_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup calls to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the positions of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup call will skip or duplicate entries.

obj_type The UUID of the type being looked up.

lookup_handle

A position in the database.

On input, the *lookup_handle* indicates the position in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

lb_$lookup_type

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found *max_results* matching entries before it reached the end of the database.

max_results

The maximum number of entries that can be returned by a single call. This should be the number of elements in the *results* array.

num_results

The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specified by the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status The completion status.

EXAMPLE

The following statement looks up information in the GLB database about the type identified by **array_id**:

```
lb_$lookup_type (&array_id, &lookup_handle, max_results,
    &num_results, results, &st);
```

FILES

idl/**lb.idl**

SEE ALSO

lb_$lookup_interface, lb_$lookup_object, lb_$lookup_range

NAME

lb_$register – register an object and an interface with the Location Broker

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$register(
        uuid_$t *object,
        uuid_$t *obj_type,
        uuid_$t *obj_interface,
        lb_$server_flag_t flags,
        unsigned char annotation[64],
        socket_$addr_t *location,
        unsigned long location_length,
        lb_$entry_t *entry,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$register(
        in object: uuid_$t;
        in obj_type: uuid_$t;
        in obj_interface: uuid_$t;
        in flags: lb_$server_flag_t;
        in annotation: array [0..63] of char;
        in location: socket_$addr_t;
        in location_length: unsigned32;
        out entry: lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$register** call registers with the Location Broker an interface to an object and the location of a server that exports that interface. This call replaces any existing entry in the Location Broker database that matches *object*, *obj_type*, *obj_interface*, and both the address family and host in *location*; if no such entry exists, the call adds a new entry to the database.

If the *flags* parameter is **lb_$server_flag_local**, the entry is registered only in the LLB database at the host where the call is issued. Otherwise, the entry is registered in both the LLB and the GLB databases.

object The UUID of the object being registered.

obj_type The UUID of the type of the object being registered.

obj_interface
The UUID of the interface being registered.

flags Must be either **lb_$server_flag_local** (specifying registration with only the LLB at the local host) or 0 (specifying registration with both the LLB and the GLB).

annotation
A character array used only for informational purposes. This field can contain a textual description of the object and the interface. For proper display by the **lb_admin** tool, the *annotation* should be terminated by a null character.

location The socket address of the server.

location_length
The length, in bytes, of the socket address specified by the *location* field.

entry A copy of the entry that was entered in the Location Broker database.

status The completion status.

EXAMPLE

The following statement from *examples*/**stacks/server.c**, the server for the **stacks** example, registers an object and an interface for array-based stacks with the Location Broker:

```
lb_$register(&astack, &astackt, &stacks_v1$if_spec.id, 0L,
    (ndr_$char *)"astack example", &loc, llen, &lb_entry[0], &st);
```

FILES

idl/**lb.idl**

SEE ALSO

lb_$unregister

NAME

lb_$unregister – remove an entry from the Location Broker database

SYNOPSIS (C)

```
#include <idl/c/lb.h>

void lb_$unregister(
        lb_$entry_t *entry,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/lb.ins.pas'

procedure lb_$unregister(
        out entry: lb_$entry_t;
        out status: status_$t);
```

DESCRIPTION

The **lb_$unregister** call removes from the Location Broker database the entry that matches *entry*. The value of *entry* should be identical to that returned by the **lb_$register** call when the database entry was created. However, **lb_$unregister** does not compare all of the fields in *entry*; it ignores the **flags** field, the **annotation** field, and the port number in the **saddr** field.

This call removes the specified entry from the LLB database on the local host (the host that issues the call). If the **flags** field of the entry is not **lb_$server_flag_local**, the call also removes the entry from the GLB database.

entry The entry being removed from the Location Broker database.

status The completion status.

EXAMPLE

The following statement from *examples*/**stacks/server.c**, the server for the **stacks** example, unregisters the first entry in the **lb_entry** array with the Location Broker:

```
lb_$unregister(&lb_entry[0], &stat);
```

FILES

***idl*/lb.idl**

SEE ALSO

lb_$register

Chapter 11

pfm_$ Calls

Contents

Annexe A.5 :

Process Fault Manager (PFM)

NAME

pfm_$intro – fault management

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ppfm.ins.pas';
```

DESCRIPTION

The **pfm_$** calls allow programs to manage signals, faults, and exceptions by establishing cleanup handlers.

We supply with the NCS software products a portable subset of the Apollo Domain/OS **pfm_$** calls.

Cleanup Handlers

A cleanup handler is a piece of code that allows a program to terminate gracefully when it receives an error. A cleanup handler begins with a **pfm_$cleanup** call and usually ends with a call to **pfm_$signal** or **pgm_$exit**, though it can also simply continue back into the program after the cleanup code.

Include Files in NCS Software

This section describes the include files for the **pfm_** interface that we provide with NCS software.

In Version 1.1 of NCK and NIDL, we supplied a **pfm.h** include file that supports the **std_$call** calling convention of Apollo SR9 system software, whereby all parameters of a call are passed by reference rather than by value. For example, a call in C source code to **pfm_$reset_cleanup** looks like

```
pfm_$reset_cleanup (crec, st)
```

even though both crec and st are passed by reference to the implementation of **pfm_$reset_cleanup**. On Apollo SR9 systems, the C compiler treats these parameters as though each were preceded by the address operator **&**. On SunOS, ULTRIX, and VMS systems with Version 1.1 of NCK or NIDL, the **pfm.h** that we supplied defines macros that convert these parameters to &crec and &st.

In Version 1.5.1 of NCK and NIDL, we supply a new include file for the **pfm_$** calls, **ppfm.h**. This is the include file for the "portable PFM" interface, an interface in the style of ANSI C. When an application invokes a call through this interface, all output parameters must be preceded by an explicit &. For example, a call to **pfm_$reset_cleanup** looks like

```
pfm_$reset_cleanup (&crec, &st)
```

since crec and st are output parameters passed by reference. This calling convention is more natural to most C programmers.

The previous include file, **pfm.h**, is still available, providing backward compatibility for programs coded according to the std_$call convention. However, we recommend that new programs include **ppfm.h**.

Include Files in Apollo SR10 Domain/OS Software

In Apollo SR10 system software, we supply the include file <apollo/pfm.h>, which defines the **pfm_** interface in the style of ANSI C.

Beginning at SR10.2, we also supply the file <apollo/ppfm.h>, which includes <apollo/pfm.h>; /usr/include/ppfm.h is a symbolic link pointing to /usr/include/apollo/ppfm.h.

Thus, you can use the directive

```
#include <ppfm.h>
```

both on Apollo SR10.2 systems and on other systems with Version 1.5.1 of NCK or NIDL.

The signatures for **pfm_$reset_cleanup** and **pfm_$rls_cleanup** in the SR10.0 and SR10.1 versions of <apollo/pfm.h> are incorrect. They have been corrected at SR10.2. These corrections may require you to modify an application developed on SR10.0 and SR10.1 Apollo systems in order to compile it on an SR10.2 Apollo system. See the reference descriptions of these calls for details.

CONSTANTS

pfm_$init_signal_handlers
: A constant used as the *flags* parameter to **pfm_$init**, causing C signals to be intercepted and converted to PFM signals.

DATA TYPES

pfm_$cleanup_rec
: An opaque data type for passing process context among cleanup handler calls.

status_$t
: A status code. Most of the NCS calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 mode : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

mode This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

pfm_$intro

STATUS CODES

pfm_$bad_rls_order
Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found
There is no pending cleanup handler.

pfm_$cleanup_set
A cleanup handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid cleanup record to a call.

pfm_$no_space
Cannot allocate storage for a cleanup handler.

status_$ok
The call was successful.

pfm_$cleanup

NAME

pfm_$cleanup – establish a cleanup handler

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

status_$t pfm_$cleanup(
        pfm_$cleanup_rec *cleanup_record)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ppfm.ins.pas';

function pfm_$cleanup(
        out cleanup_record: pfm_$cleanup_rec): status_$t;
```

DESCRIPTION

The **pfm_$cleanup** call establishes a cleanup handler that is executed when a fault occurs. A cleanup handler is a piece of code executed before a program exits when a signal is received by the process. The cleanup handler begins where **pfm_$cleanup** is called; the **pfm_$cleanup** call registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to **pfm_$cleanup**.

There can be more than one cleanup handler in a program. Multiple cleanup handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established cleanup handler and ending with the first cleanup handler.

On Apollo systems, a default cleanup handler is established at program invocation. The default cleanup handler is always called last, just before a program exits, and releases any system resources still held before returning control to the process that invoked the program.

On other systems, there is no default cleanup handler.

When called to establish a cleanup handler, **pfm_$cleanup** returns the status **pfm_$cleanup_set** to indicate that the cleanup handler was successfully established. When the cleanup handler is entered in response to a fault signal, **pfm_$cleanup** effectively returns the value of the fault that triggered the cleanup handler.

See the reference description of **pfm_$init** for a list of the C signals that the PFM package intercepts.

cleanup_record
A record of the context when **pfm_$cleanup** is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by **pfm_$rls_cleanup** and **pfm_$reset_cleanup** to restore the context of the calling process at the cleanup handler entry point.

NOTE

The **pfm_$cleanup** call implicitly performs a **pfm_$inhibit**. Cleanup handler code hence runs with asynchronous faults inhibited. When **pfm_$cleanup** returns something other than **pfm_$cleanup_set**, indicating that a fault has occurred, there are four possible ways to leave the cleanup code:

- The program can call **pfm_$signal** to start the next cleanup handler with a fault signal you specify.
- The program can call **pgm_$exit** to start the next cleanup handler with a status of **status_$ok**.
- The program can continue with the code following the cleanup handler. It should generally call **pfm_$enable** to re-enable asynchronous faults. Execution continues from the end of the cleanup handler code; it does not resume where the fault signal was received.
- The program can re-establish the cleanup handler by calling **pfm_$reset_cleanup** (which implicitly performs a **pfm_$enable**) before proceeding.

SEE ALSO

pfm_$init, pfm_$signal

NAME

pfm_$enable – enable asynchronous faults

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$enable(void)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$enable;
```

DESCRIPTION

The **pfm_$enable** call enables asynchronous faults after they have been inhibited by a call to **pfm_$inhibit**; **pfm_$enable** causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when **pfm_$enable** returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to **pfm_$inhibit** and **pfm_$enable**, the process receives the first asynchronous fault received while faults were inhibited.

SEE ALSO

pfm_$enable_faults, pfm_$inhibit

pfm_$enable_faults

NAME

pfm_$enable_faults – enable asynchronous faults

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$enable_faults(void)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$enable_faults;
```

DESCRIPTION

The **pfm_$enable_faults** call enables asynchronous faults after they have been inhibited by a call to **pfm_$inhibit_faults**; **pfm_$enable_faults** causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when **pfm_$enable_faults** returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to **pfm_$inhibit_faults** and **pfm_$enable_faults**, the process receives the first asynchronous fault received while faults were inhibited.

SEE ALSO

pfm_$enable, pfm_$inhibit_faults

pfm_$inhibit

NAME

pfm_$inhibit – inhibit asynchronous faults

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$inhibit(void);
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$inhibit;
```

DESCRIPTION

The **pfm_$inhibit** call prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to **pfm_$inhibit** can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

On systems with Concurrent Programming Support (CPS), **pfm_$inhibit** also disables time-sliced task switching. It does not prevent task switching due to voluntary task yielding, either explicitly via **task_$yield** or implicitly via other functions that yield. You should not use **pfm_$inhibit** for critical region concurrency control; use the **mutex_** facility instead.

See the reference description of **pfm_$init** for a list of the C signals that the PFM package intercepts.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

SEE ALSO

pfm_$enable, pfm_$inhibit_faults, pfm_$init
Concurrent Programming Support Reference

pfm_$inhibit_faults

NAME

pfm_$inhibit_faults – inhibit asynchronous faults but allow time-sliced task switching

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$inhibit_faults(void);
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$inhibit_faults;
```

DESCRIPTION

The **pfm_$inhibit_faults** call prevents asynchronous faults, except for time-sliced task switching, from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to **pfm_$inhibit_faults** can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

See the reference description of **pfm_$init** for a list of the C signals that the PFM package intercepts.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

SEE ALSO

pfm_$enable_faults, pfm_$inhibit, pfm_$init

pfm_$init

NAME

pfm_$init – initialize the PFM package

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$init(
        unsigned long flags)
```

DESCRIPTION

The **pfm_$init** call initializes the PFM package. The *flags* parameter indicates which initialization activities to perform.

flags Currently, only one value is valid:

pfm_$init_signal_handlers
This causes C signals to be intercepted and converted to PFM signals. On UNIX and VMS systems, the signals intercepted are SIGINT, SIGILL, SIGFPE, SIGTERM, SIGHUP, SIGQUIT, SIGTRAP, SIGBUS, SIGSEGV, and SIGSYS. On MS-DOS systems, the first four of these, plus SIGABRT, are intercepted.

On Apollo systems, the PFM package does not require initialization, and **pfm_$init** is a no-op. On all other systems, applications that use the PFM package should invoke **pfm_$init** before invoking any other NCS calls.

NAME

pfm_$reset_cleanup – reset a cleanup handler

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$reset_cleanup(
        pfm_$cleanup_rec *cleanup_record,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$reset_cleanup(
        in cleanup_record: pfm_$cleanup_rec;
        out status: status_$t);
```

DESCRIPTION

The **pfm_$reset_cleanup** call re-establishes the cleanup handler last entered so that any subsequent errors enter it first. This procedure should only be used within cleanup handler code.

A **pfm_$reset_cleanup** implicitly performs a **pfm_$enable**, thereby undoing the implicit **pfm_$inhibit** that **pfm_$cleanup** performs.

cleanup_record
: A record of the context at the cleanup handler entry point. It is supplied by **pfm_$cleanup** when the cleanup handler is first established.

status
: The completion status.

NOTE

This note concerns use of **pfm_$reset_cleanup** on Apollo systems.

In the SR10.0 and SR10.1 versions of **<apollo/pfm.h>**, the first argument of **pfm_$reset_cleanup** is incorrectly preceded by an ampersand (&). In the SR10.2 version, the first argument is correctly preceded by an asterisk (*).

Programs compiled under SR10.0 or SR10.1 will continue to run correctly, since the implementation of **pfm_$reset_cleanup** has not changed, but you may need to modify these programs in order to compile them under SR10.2. Invocations of **pfm_$reset_cleanup** that looked like

```
pfm_$reset_cleanup(crec, &st)
```

when compiled under SR10.0 and SR10.1 must be modified to

```
pfm_$reset_cleanup(&crec, &st)
```

when compiled under SR10.2.

NAME

pfm_$rls_cleanup – release a cleanup handler

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$rls_cleanup(
        pfm_$cleanup_rec *cleanup_record,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$rls_cleanup(
        in cleanup_record: pfm_$cleanup_rec;
        out status: status_$t);
```

DESCRIPTION

The **pfm_$rls_cleanup** call releases the cleanup handler associated with *cleanup_record*.

On Apollo systems, this call releases the specified cleanup handler and all cleanup handlers established after it.

On other systems, this call releases only the specified cleanup handler, and only the most recently established cleanup handler can be released.

If you are concerned about portability, use **pfm_$rls_cleanup** only to release the most recent cleanup handler.

cleanup_record
: The cleanup record for the cleanup handler to release.

status
: The completion status.

ERRORS

pfm_$bad_rls_order
: The caller attempted to release a cleanup handler other than the one most recently established. On Apollo systems, this status is only a warning; the specified cleanup handler is released, along with any established after it. On other systems, this status probably indicates a user programming error; no cleanup handlers are released, and continued execution may result in more serious errors.

pfm_$rls_cleanup

NOTE

This note concerns use of **pfm_$rls_cleanup** on Apollo systems.

In the SR10.0 and SR10.1 versions of **<apollo/pfm.h>**, the first argument of **pfm_$rls_cleanup** is incorrectly preceded by an ampersand (&). In the SR10.2 version, the first argument is correctly preceded by an asterisk (*).

Programs compiled under SR10.0 or SR10.1 will continue to run correctly, since the implementation of **pfm_$rls_cleanup** has not changed, but you may need to modify these programs in order to compile them under SR10.2. Invocations of **pfm_$rls_cleanup** that looked like

```
pfm_$rls_cleanup(crec, &st)
```

when compiled under SR10.0 and SR10.1 must be modified to

```
pfm_$rls_cleanup(&crec, &st)
```

when compiled under SR10.2.

pfm_$signal

NAME

pfm_$signal – signal the calling process

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$signal(status_$t fault_signal)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$signal(in fault_signal: status_$t);
```

DESCRIPTION

The **pfm_$signal** call signals the fault specified by *fault_signal* to the calling process. It is usually called to leave cleanup handlers.

fault_signal
: A fault code.

NOTE

This call does not return when successful.

Annexe A.6 :

RPC RunTime

Chapter 12

pgm_$ Calls

Contents

pgm_$intro

NAME

pgm_$intro – program management

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
```

DESCRIPTION

We supply with the NCS software products a portable version of the Apollo Domain/OS pgm_$exit call. The include file for the "portable PFM" interface (see Chapter 11) contains a declaration for this call.

pgm_$exit

NAME

pgm_$exit – exit a program

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pgm_$exit(void)
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$exit;
```

DESCRIPTION

The pgm_$exit call exits from the calling program.

Any cleanup handlers that have been established are executed in order from the most recently established to the first.

On Apollo systems, this call invokes pfm_$signal with a fault code equal to the last severity level set by pgm_$set_severity, or pgm_$ok if pgm_$set_severity was not called.

On other systems, this call invokes pfm_$signal with a fault code of status_$ok.

SEE ALSO

pfm_$cleanup, pfm_$signal

Chapter 13

rpc_$ Calls

Contents

rpc_$intro

NAME

rpc_$intro – interface to the Remote Procedure Call runtime library

SYNOPSIS (C)

```
#include <idl/c/rpc.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/rpc.ins.pas';
```

DESCRIPTION

The rpc_$ calls implement the NCS Remote Procedure Call (RPC) mechanism.

The rpc_ interface is defined by the file *idl*/rpc.idl, where the symbol *idl* denotes the system idl directory. On Apollo workstations and other UNIX systems, *idl* is /usr/include/idl.

Most of the rpc_$ calls can be used only by clients or only by servers. We indicate this aspect of their usage at the beginning of each call description, in the "NAME" section.

EXTERNAL VARIABLES

The following external variable is used in rpc_$ calls:

uuid_$nil
An external uuid_$t variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

CONSTANTS

The following constants are used in rpc_$ calls:

rpc_$mod
A module code indicating the RPC module. See the description of the status_$t type.

rpc_$unbound_port
A port number indicating that no port is specified. This constant is obsolete; use instead socket_$unspec_port.

The following 16-bit-integer constants are used to specify the address families in socket_$addr_t structures. Note that several of the rpc_$ and socket_$ calls use the 32-bit-integer equivalents of these values.

socket_$unspec
Address family is unspecified.

socket_$internet
Internet Protocols (IP).

socket_$dds
Domain protocols (DDS).

DATA TYPES

The following data types are used in rpc_$ calls:

handle_t An RPC handle.

rpc_$epv_t
An entry point vector (EPV). An array of pointers to server stub routines.

rpc_$generic_epv_t
An entry point vector. An array of pointers to generic server stub routines.

rpc_$intro

rpc_$if_spec_t
An RPC interface specifier. An opaque structure containing information about an interface, including the UUID, the version number, the number of operations in the interface, and any well-known ports used by servers that export the interface. Applications may need to access two members of rpc_$if_spec_t:

id A uuid_$t indicating the interface UUID.

vers An unsigned 32-bit integer indicating the interface version.

rpc_$mgr_epv_t
An entry point vector. An array of pointers to manager routines.

rpc_$shut_check_fn_t
A pointer to a function. If a server supplies this pointer to rpc_$allow_remote_shutdown, the function will be called when a remote shutdown request arrives, and if the function returns true, the shutdown is allowed. The following C definition for rpc_$shut_check_fn_t illustrates the syntax for this function:

```
typedef boolean (*rpc_$shut_check_fn_t) (
    handle_t h,
    status_$t *st);
```

The handle argument can be used to determine information about the remote caller.

socket_$addr_t
A socket address record that uniquely identifies a socket. See the socket_$intro section of Chapter 15 for a full description of this type.

status_$t A status code. Most of the NCS calls supply their completion status in this format. The status_$t type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use status_$t as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If all bits are zero, status_$ok, the call that supplied the status was successful.

fail If this bit is set, the error was not in the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

mode This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. See the **uuid_$intro** section of Chapter 16 for a full description of this type.

STATUS CODES

The following are status codes returned by rpc_$ calls:

rpc_$addr_in_use
The address and port specified in an rpc_$use_family_wk call are already in use. This is caused by multiple calls to rpc_$use_family_wk for the same well-known port.

rpc_$bad_pkt
The server or client has received an ill-formed packet.

rpc_$cant_bind_sock
The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$comm_failure
The client was unable to get a response from the server.

rpc_$illegal_register
You are trying to register an interface that is already registered, and you are using an EPV (with the rpc_$register call) or a generic EPV (with the rpc_$register_mgr call) different from the one used when the interface was first registered.

rpc_$not_in_call
An internal error.

rpc_$op_rng_error
The requested operation does not correspond to a valid operation on the requested interface.

rpc_$proto_error
An internal protocol error.

rpc_$too_many_ifs
The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called rpc_$use_family or rpc_$use_family_wk too many times.

rpc_$unbound_handle
The handle is not bound and does not represent a particular host address. Returned by **rpc_$inq_binding**.

rpc_$unk_if
The requested interface is not known. The server has not registered the interface with the RPC runtime library, the version number in the request does not match the version number of the registered interface, or the UUID in the request does not match the UUID of the registered interface.

rpc_$wrong_boot_time
The server boot time value maintained by the client does not correspond to the current server boot time. The server program was probably restarted while the client program was running.

rpc_$you_crashed
This error can occur if a server has crashed and restarted. The RPC runtime library at a client host sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

status_$ok
The call was successful.

FILES

***idl*/rpc.idl**

NAME

rpc_$alloc_handle – create an RPC handle (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

handle_t rpc_$alloc_handle(
        uuid_$t *object,
        unsigned long family,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

function rpc_$alloc_handle(
        in object: uuid_$t;
        in family: unsigned32;
        out status: status_$t): handle_t;
```

DESCRIPTION

The **rpc_$alloc_handle** call creates an unbound RPC handle, a handle that identifies a particular object but not a particular server or host.

When a client uses an unbound handle to make a remote procedure call, the RPC runtime library broadcasts the call on the local network. If a well-known port was specified in the definition of the requested interface, the call is broadcast to that port. Otherwise, the call is broadcast to the Local Location Broker (LLB) forwarding port. The client RPC runtime library returns the first response that it receives and binds the handle to the server.

Because broadcasting is inefficient, we discourage use of unbound handles.

object The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.

family The address family to use in communications to access the object. The **rpc_$intro** section describes possible values.

status The completion status.

EXAMPLE

The following statement allocates a handle that identifies a print queue object:

```
h = rpc_$alloc_handle (&printq_id, (unsigned long) socket_$dds, &st);
```

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$dup_handle, rpc_$free_handle, rpc_$set_binding

NAME

rpc_$allow_remote_shutdown – allow or disallow remote shutdown of a server (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$allow_remote_shutdown(
        unsigned long allow,
        rpc_$shut_check_fn_t checkproc,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$allow_remote_shutdown(
        in allow: unsigned32;
        in checkproc: rpc_$shut_check_fn_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$allow_remote_shutdown** call allows or disallows remote callers to shut down a server via **rrpc_$shutdown**.

By default, servers do not allow remote shutdown. If a server calls **rpc_$allow_remote_shutdown** with *allow* "true" (nonzero) and *checkproc* nil, then remote shutdown will be allowed. If *allow* is "true" and *checkproc* is not nil, then when a remote shutdown request arrives, the function referenced by *checkproc* is called and the shutdown is allowed if the function returns true. If *allow* is "false" (zero), remote shutdown is disallowed.

allow A value indicating "false" if zero, "true" otherwise.

checkproc
A pointer to a Boolean function. See the **rpc_$intro** section for a full description of **rpc_$shut_check_fn_t**.

status The completion status.

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$shutdown, rrpc_$shutdown

NAME

rpc_$bind – allocate an RPC handle and set its binding to a server (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

handle_t rpc_$bind(
        uuid_$t *object,
        socket_$addr_t *sockaddr,
        unsigned long slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

function rpc_$bind(
        in object: uuid_$t;
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out status: status_$t): handle_t;
```

DESCRIPTION

The **rpc_$bind** function creates an RPC handle that represents the object identified by *object*. This call is equivalent to an **rpc_$alloc_handle** call followed by an **rpc_$set_binding** call.

If you supply a fully specified *sockaddr*, **rpc_$bind** creates a fully bound handle, one whose location information identifies a particular port at a particular host. When a client uses a fully bound handle to make a remote procedure call, the RPC runtime library delivers the call directly to the host and port identified in the handle.

If the port number in *sockaddr* is **socket_$unspec_port**, **rpc_$bind** creates a bound-to-host handle, one whose location information identifies a host, but not a port. When a client uses a bound-to-host handle to make a remote procedure call, the RPC runtime library sends the call to the host identified in the handle; unless the requested interface specifies a well-known port, the call is sent to the Local Location Broker (LLB) forwarding port, and the LLB forwards the call to the server.

object The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.

sockaddr The socket address of the server.

slength The length, in bytes, of *sockaddr*.

status The completion status.

EXAMPLE

The following statement from *examples/binoplu/client.c*, the **binoplu** example client program, binds the client to the specified object and socket address. The **entry** is from the results of a previous Location Broker lookup call.

```
h = rpc_$bind (&uuid_$nil, &entry.saddr, entry.saddr_len, &st);
```


FILES

idl/rpc.idl

SEE ALSO

rpc_$clear_binding, rpc_$clear_server_binding, rpc_$set_binding

NAME

rpc_$clear_binding – unset the binding of an RPC handle (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$clear_binding(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$clear_binding(
        in handle: handle_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$clear_binding** call clears the binding of an RPC handle to a particular server and host. The handle continues to represent an object but is unbound. It can be reused to access the object, either by broadcasting or after resetting the binding to another server.

When a client uses an unbound handle to make a remote procedure call, the RPC runtime library broadcasts the call on the local network. If a well-known port was specified in the definition of the requested interface, the call is broadcast to that port. Otherwise, the call is broadcast to the Local Location Broker (LLB) forwarding port. The client RPC runtime library returns the first response that it receives and binds the handle to the server.

Because broadcasting is inefficient, we discourage use of unbound handles.

The **rpc_$clear_binding** call is the inverse of the **rpc_$set_binding** call.

handle The RPC handle whose binding is being cleared.

status The completion status.

EXAMPLE

The following statement clears the binding represented in **handle**:

```
rpc_$clear_binding (handle, &st);
```

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$bind, rpc_$clear_server_binding, rpc_$set_binding

NAME

rpc_$clear_server_binding – unset the binding of an RPC handle to a server (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$clear_server_binding(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$clear_server_binding(
        in handle: handle_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$clear_server_binding** call clears the binding of an RPC handle to a particular server (that is, a particular port number), but does not clear the binding to a host (that is, a network address). The handle continues to represent an object.

This call replaces a fully bound handle with a bound-to-host handle. A bound-to-host handle identifies an object and a host but does not identify a server at that host.

When a client uses a bound-to-host handle to make a remote procedure call, the RPC runtime library sends the call to the host identified in the handle. If a well-known port was specified in the definition of the requested interface, the call is delivered to that port. Otherwise, the call is delivered to the Local Location Broker (LLB) forwarding port. The LLB, provided a server for the requested object and interface has registered with it, forwards the call to the port on which the server is listening. When the remote procedure call returns, the RPC runtime library at the client host binds the handle to that port, and any subsequent calls are sent directly to the server.

The **rpc_$clear_server_binding** call is useful for client error recovery when a server that listens on opaque ports dies and restarts; the restarted server may be listening on a different port. This call enables the client to unbind from the old port while retaining the binding to the host, so that when the client sends its next request, the call is forwarded to the new port.

handle The RPC handle whose server binding is being cleared.

status The completion status.

EXAMPLE

The following statement clears the server binding represented in **handle**:

```
rpc_$clear_server_binding (handle, &st);
```

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$bind, rpc_$clear_binding, rpc_$set_binding

rpc_$dup_handle

NAME

rpc_$dup_handle – make a copy of an RPC handle (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

handle_t rpc_$dup_handle(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

function rpc_$dup_handle(
        in handle: handle_t;
        out status: status_$t): handle_t;
```

DESCRIPTION

The **rpc_$dup_handle** call returns a copy of an existing RPC handle. Both handles can then be used in the client program for concurrent multiple accesses to a binding. Because all duplicates of a handle reference the same data, an **rpc_$set_binding**, **rpc_$clear_binding**, or **rpc_$clear_server_binding** call made on any one duplicate affects all duplicates. However, an RPC handle is not freed until **rpc_$free_handle** is called on all copies of the handle.

The **rpc_$dup_handle** call is designed to support programs that use Concurrent Programming Support (CPS). It allows multiple threads of execution within a process to use separate copies of the handle but to share the resources that are identified by the handle.

handle The RPC handle to be copied.

status The completion status.

EXAMPLE

The following statement creates as **handle2** a copy of **handle1**:

```
handle2 = rpc_$dup_handle (handle1, &st);
```

FILES

idl/rpc.idl

SEE ALSO

rpc_$alloc_handle, rpc_$free_handle

rpc_$free_handle

NAME

rpc_$free_handle – free an RPC handle (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$free_handle(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$free_handle(
        in handle: handle_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$free_handle** call frees an RPC handle. This call releases the resources identified by the RPC handle. The client program cannot use a handle after it is freed.

If you make several remote procedure calls that access the same object but at different locations, it is more efficient to use **rpc_$set_binding** (replacing the binding in an existing handle) than to use **rpc_$free_handle** and **rpc_$bind** (creating a new handle).

If you create copies of an RPC handle by using the **rpc_$dup_handle** call, the handles are not freed until **rpc_$free_handle** is called once for each copy of the handle. However, the RPC runtime library does not differentiate between calling **rpc_$free_handle** several times on one copy of a handle and calling it one time for each of several copies of a handle. Therefore, if you use duplicate handles, you must ensure that no thread inadvertently makes multiple **rpc_$free_handle** calls on a single handle.

handle The RPC handle to be freed.

status The completion status.

EXAMPLE

The following statements free two copies of a handle:

```
rpc_$free_handle (handle1, &st);
rpc_$free_handle (handle2, &st);
```

FILES

idl/rpc.idl

SEE ALSO

rpc_$alloc_handle, rpc_$dup_handle

NAME

rpc_$inq_binding – return the socket address represented by an RPC handle (CLIENT OR SERVER)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$inq_binding(
        handle_t handle,
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$inq_binding(
        in handle: handle_t;
        out sockaddr: socket_$addr_t;
        out slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$inq_binding** call enables a client or server to determine the socket address identified by an RPC handle.

A client might invoke **rpc_$inq_binding** if it has used an unbound handle to make a remote procedure call and it wants to determine the particular server that responded to the call.

Conversely, a server might use **rpc_$inq_binding** to identify its clients. The RPC runtime library manipulates the location information in an RPC handle so that on the server side of an application, the handle specifies the location of the client making the call.

handle	An RPC handle.
sockaddr	The socket address represented by *handle*.
slength	The length, in bytes, of *sockaddr*.
status	The completion status.

ERRORS

rpc_$unbound_handle
The handle is not bound and does not represent a specific host address.

EXAMPLE

The Location Broker administrative tool, **lb_admin**, uses the following statement to determine the particular Global Location Broker (GLB) that responded to a lookup request:

```
rpc_$inq_binding (glb_$handle, &global_broker_addr,
    &global_broker_addr_len, &status);
```


FILES

idl/rpc.idl

SEE ALSO

rpc_$bind, rpc_$set_binding

rpc_$inq_object

NAME

rpc_$inq_object – return the object UUID represented by an RPC handle (CLIENT OR SERVER)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$inq_object(
        handle_t handle,
        uuid_$t *object,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$inq_object(
        in handle: handle_t;
        out object: uuid_$t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$inq_object** call enables a client or server to determine the particular object that a handle represents.

If a server exports an interface through which clients can access several objects, it can use **rpc_$inq_object** to determine the object requested in a call. This call requires an RPC handle as input, so the server can make the call only if the interface uses explicit handles, that is, if each operation in the interface has a handle parameter.

handle An RPC handle.

object The UUID of the object identified by *handle*.

status The completion status.

EXAMPLE

A database server that manages several databases must determine the particular database to be accessed when it receives a remote procedure call. Each manager routine therefore makes the following call; the routine then uses the returned UUID to identify the database to be accessed.

```
rpc_$inq_object (handle, &db_uuid, &st);
```

FILES

idl/**rpc.idl**

rpc_$listen

NAME

rpc_$listen – listen for and handle remote procedure call packets (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$listen(
        unsigned long max_calls,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$listen(
        in max_calls: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$listen** call dispatches incoming remote procedure call requests to manager procedures and returns the responses to the client. You must issue **rpc_$use_family** or **rpc_$use_family_wk** before you use **rpc_$listen**. This call normally does not return.

On systems that have Concurrent Programming Support (CPS), the RPC runtime library at the server host can use CPS to handle several requests simultaneously. The *max_calls* parameter specifies how many concurrent requests are allowed; the RPC runtime library supports at most 10 concurrent requests; If a server uses concurrency, all manager routines must be re-entrant; that is, they must maintain concurrency controls on any non-local variables to prevent conflicts among the various threads of execution.

On systems that do not have CPS, *max_calls* is ignored, and the server processes only one call at a time.

max_calls

The maximum number of calls that the server is allowed to process concurrently. Regardless of the value for *max_calls*, systems without CPS support only one process, and systems with CPS support at most 10 concurrent processes.

status The completion status.

EXAMPLE

The following statement listens for incoming remote procedure call requests, handling up to five concurrently:

```
rpc_$listen (5, &status);
```

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$shutdown

NAME

rpc_$name_to_sockaddr – convert a host name and port number to a socket address (CLIENT OR SERVER)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$name_to_sockaddr(
        unsigned char name[256],
        unsigned long nlength,
        unsigned long port,
        unsigned long family,
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$name_to_sockaddr(
        in name: array [0..255] of char;
        in nlength: unsigned32;
        in port: unsigned32;
        in family: unsigned32;
        out sockaddr: socket_$addr_t;
        out slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

This call is obsolete. See the note below.

The **rpc_$name_to_sockaddr** call provides the socket address for a socket, given the host name, the port number, and the address family.

You can specify socket address information either by passing one text string in the *name* parameter or by passing each of the three elements in a separate parameter. In the latter case, the *name* parameter should contain only the host name.

name A string that contains a host name and, optionally, a port and an address family. The format is *family*:*host*[*port*], where *family*: and [*port*] are optional. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter. The *family* can be either **dds** or **ip**; *host* is the host name; *port* is an integer port number.

nlength The number of characters in *name*.

port The socket port number. This parameter should have the value **socket_$unspec_port** if you are not specifying a well-known port; in this case, the returned socket address will specify the Local Location Broker (LLB) forwarding port at *host*. If you specify the port number in the *name* parameter, this parameter is ignored.

family The address family to use for the socket address. This value corresponds to the communications protocol used to access the socket and determines how the *sockaddr* is expressed. The **rpc_$intro** section describes possible values. If you specify the address family in the *name* parameter, this parameter must have the value **socket_$unspec**.

sockaddr The socket address corresponding to *name*, *port*, and *family*.

slength The length, in bytes, of *sockaddr*.

status The completion status.

NOTE

This call has been superseded by the **socket_$from_name** call.

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$sockaddr_to_name, socket_$from_name

NAME

rpc_$register – register an interface (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$register(
        rpc_$if_spec_t *ifspec,
        rpc_$epv_t epv,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$register(
        in ifspec: rpc_$if_spec_t;
        in epv: rpc_$epv_t;
        out status: status_$t);
```

DESCRIPTION

This call is obsolete. See the note below.

The rpc_$register call registers an interface with the RPC runtime library. After an interface is registered, the RPC runtime library will dispatch requests for that interface to the server.

You can call rpc_$register several times with the same interface (for example, from various subroutines of the same server), but each call must specify the same EPV. Each registration increments a reference count for the registered interface; you must call rpc_$unregister an equal number of times to unregister the interface.

To generate stubs for an application whose server uses rpc_$register, specify the -s option of the NIDL Compiler when you compile the interface definition.

ifspec The interface being registered.

epv The entry point vector (EPV) for the operations in the interface. The EPV is normally defined in the server stub that is generated by the NIDL Compiler from an interface definition.

status The completion status.

ERRORS

rpc_$too_many_ifs
The maximum number of interfaces is already registered with the server.

rpc_$illegal_register
You are trying to register an interface that is already registered, and you are specifying an EPV different from the one specified when the interface was first registered.

NOTE

This call has been superseded by the rpc_$register_mgr and rpc_$register_object calls, which enable a server to export more than one version of an interface and to implement an interface for more than one type.

To generate stubs for an application whose server uses rpc_$register_mgr and rpc_$register_object, specify the -m option of the NIDL Compiler when you compile the interface definition.

FILES

idl/rpc.idl

SEE ALSO

rpc_$register_mgr, rpc_$register_object, rpc_$unregister

rpc_$register_mgr

NAME

rpc_$register_mgr – register a manager (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$register_mgr(
        uuid_$t *type,
        rpc_$if_spec_t *ifspec,
        rpc_$generic_epv_t sepv,
        rpc_$mgr_epv_t mepv,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$register_mgr(
        in type: uuid_$t;
        in ifspec: rpc_$if_spec_t;
        in sepv: rpc_$generic_epv_t;
        in mepv: rpc_$mgr_epv_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$register_mgr** call registers the set of manager procedures that implement a specified interface for a specified type.

This call and **rpc_$register_object** supersede the obsolete **rpc_$register** call.

Servers can invoke this call several times with the same interface (*ifspec*) and generic EPV (*sepv*) but with a different object type (*type*) and manager EPV (*mepv*) on each invocation. This technique allows a server to export several implementations of the same interface.

Servers that export several versions of the same interface (but not different implementations for different types) must also use **rpc_$register_mgr**, not **rpc_$register**. Such servers can supply **uuid_$nil** as the *type* to **rpc_$register_mgr**.

If a server uses **rpc_$register_mgr** to register a manager for a specific interface and a specific non-nil type, the server must use **rpc_$register_object** to register an object.

To generate stubs for an application whose server uses **rpc_$register_mgr** and **rpc_$register_object**, specify the **–m** option of the NIDL Compiler when you compile the interface definition.

type The UUID of the type being registered.

ifspec The interface being registered.

sepv The generic EPV, a vector of pointers to server stub procedures.

mepv The manager EPV, a vector of pointers to manager procedures.

status The completion status.

rpc_$register_mgr

EXAMPLE

The following statement from *examples/stacks/server.c*, the **stacks** example server program, registers a manager for array-based stacks:

```
rpc_$register_mgr (&astackt, &stacks_v1$if_spec, stacks_v1$server_epv,
    (rpc_$mgr_epv_t)&stacks_v1$amanager_epv, &st);
```

FILES

idl/rpc.idl

SEE ALSO

rpc_$register, rpc_$register_object, rpc_$unregister

NAME

rpc_$register_object – register an object (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$register_object(
        uuid_$t *object,
        uuid_$t *type,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$register_object(
        in object: uuid_$t;
        in type: uuid_$t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$register_object** call declares that a server supports operations on a particular object and declares the type of that object.

This call and **rpc_$register_mgr** supersede the obsolete **rpc_$register** call.

A server must register objects via **rpc_$register_object** only if it registers interfaces via **rpc_$register_mgr**. When a server receives a call, the RPC runtime library searches for the object identified in the call (that is, the object that the client specified in the handle) among the objects registered by the server. If the object is found, the type of the object determines which of the manager EPVs should be used to operate on the object.

To generate stubs for an application whose server uses **rpc_$register_mgr** and **rpc_$register_object**, specify the **–m** option of the NIDL Compiler when you compile the interface definition.

object The UUID of the object being registered.

type The UUID of the type of the object.

status The completion status.

EXAMPLE

The following statement from *examples/stacks/server.c*, the stacks example server program, registers an array-based stack object:

```
rpc_$register_object (&astack, &astackt, &st);
```

FILES

*idl/*rpc.idl

SEE ALSO

rpc_$register, rpc_$register_mgr, rpc_$unregister

NAME

rpc_$set_async_ack – set or clear asynchronous-ack mode (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$set_async_ack(
        unsigned long on)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$set_async_ack(
        in on: unsigned32);
```

DESCRIPTION

The **rpc_$set_async_ack** call sets or clears asynchronous-ack mode (see the "Background" section below) in a client.

on If "true" (nonzero), asynchronous-ack mode is set. If "false" (zero), synchronous-ack mode is set.

MS-DOS Systems and Systems with CPS

On MS-DOS systems and on systems that have Concurrent Programming Support, **rpc_$set_async_ack** has no effect. These systems always use asynchronous-ack mode.

Other Systems

On other systems, synchronous-ack mode is the default. Calling **rpc_$set_async_ack** with a nonzero value for *on* sets asynchronous-ack mode. Calling it with *on* zero sets synchronous-ack mode.

Background

After a client makes a remote procedure call and receives a reply from a server, the RPC runtime library at the client acknowledges its receipt of the reply. This "reply ack" can occur either synchronously (before the runtime library returns to the caller) or asynchronously (after the runtime library returns to the caller)

It is generally good to allow asynchronous reply acks. Asynchronous-ack mode can save the client runtime library from making explicit reply acks, because after a client receives a reply, it may shortly issue another call that can act as an implicit ack.

Asynchronous-ack mode requires that an "alarm" be set to go off sometime after the remote procedure call returns. Unfortunately, setting the alarm can cause two problems: (1) There may be only one alarm that can be set, and the application itself may be trying to use it. (2) If at the time the alarm goes off the application is blocked in a UNIX system call that is doing I/O to a "slow device" (such as a terminal), the system call will return an error (with the EINTR errno); the application may not be coded to expect this error. If neither of these problems obtains, the application should set asynchronous-ack mode to get greater efficiency.

FILES

*idl/*rpc.idl

NAME

rpc_$set_binding – bind an RPC handle to a server (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$set_binding(
        handle_t handle,
        socket_$addr_t *sockaddr,
        unsigned long slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$set_binding(
        in handle: handle_t;
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$set_binding** call sets the binding of an RPC handle to the specified socket address. You can use this call either to set the binding in an unbound handle or to replace the existing binding in a fully bound or bound-to-host handle.

If you supply a fully specified *sockaddr*, *handle* becomes a fully bound handle, one whose location information identifies a particular port at a particular host. When a client uses a fully bound handle to make a remote procedure call, the RPC runtime library delivers the call directly to the host and port identified in the handle.

If the port number in *sockaddr* is **socket_$unspec_port**, *handle* becomes a bound-to-host handle, one whose location information identifies a host, but not a port. When a client uses a bound-to-host handle to make a remote procedure call, the RPC runtime library sends the call to the host identified in the handle; unless the requested interface specifies a well-known port, the call is sent to the Local Location Broker (LLB) forwarding port, and the LLB forwards the call to the server.

handle An RPC handle.

sockaddr The socket address of the server with which the handle is being associated.

slength The length, in bytes, of *sockaddr*.

status The completion status.

EXAMPLE

The following statement sets the binding on the handle h to the first server in the *lbresults* array, which was returned by a previous Location Broker lookup call:

```
rpc_$set_binding (h, &lbresults[0].saddr, lbresults[0].saddr_len, &st);
```


FILES

idl/rpc.idl

SEE ALSO

rpc_$alloc_handle, rpc_$clear_binding, rpc_$clear_server_binding

NAME

rpc_$set_fault_mode – set the fault-handling mode for a server (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

unsigned long rpc_$set_fault_mode(
        unsigned long on)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

function rpc_$set_fault_mode(
        in on: unsigned32): unsigned32;
```

DESCRIPTION

The **rpc_$set_fault_mode** function controls the handling of faults that occur in server routines.

In the default mode, the server reflects faults back to the client and continues processing. Calling **rpc_$set_fault_mode** with a nonzero value for *on* sets the fault-handling mode so that the server sends an **rpc_$comm_failure** fault back to the client and exits. (In a tasking environment on a system with Concurrent Programming Support, the distinguished task is signaled.) Calling rpc_$set_fault_mode with *on* zero resets the fault-handling mode to the default.

This function returns the previous setting of the fault-handling mode.

on If "true" (nonzero), the server exits when a fault occurs. If "false" (zero), the server reflects faults back to the client.

FILES

idl/**rpc.idl**

NAME

rpc_$set_short_timeout – set or clear short-timeout mode (CLIENT ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

unsigned long rpc_$set_short_timeout(
        handle_t handle,
        unsigned long on,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

function rpc_$set_short_timeout(
        in handle: handle_t;
        in on: unsigned32;
        out status: status_$t): unsigned32;
```

DESCRIPTION

The **rpc_$set_short_timeout** function sets or clears short-timeout mode on a handle. If a client uses a handle in short-timeout mode to make a remote procedure call, but the server shows no signs of life, the call fails quickly. As soon as the server shows signs of being alive, standard timeouts take effect and apply for the remainder of the call.

Calling **rpc_$set_short_timeout** with a nonzero value for *on* sets short-timeout mode. Calling it with *on* zero sets standard timeouts. Standard timeouts are the default.

This function returns the previous setting of the timeout mode.

handle An RPC handle.

on If "true" (nonzero), short-timeout mode is set on *handle*. If "false" (zero), standard timeouts are set.

status The completion status.

FILES

idl/**rpc.idl**

NAME

rpc_$shutdown – shut down a server (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$shutdown(
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$shutdown(
        out status: status_$t);
```

DESCRIPTION

The **rpc_$shutdown** call shuts down a server. When this call is executed, the server stops processing incoming calls, and **rpc_$listen** returns. On a system with Concurrent Programming Support, if the server is running in a tasking environment, this call kills all "listen tasks."

If **rpc_$shutdown** is called from within a remote procedure, that procedure completes and the server shuts down after replying to the caller.

status The completion status.

FILES

idl/rpc.idl

SEE ALSO

rpc_$allow_remote_shutdown, rpc_$shutdown

NAME

rpc_$sockaddr_to_name – convert a socket address to a host name and port number (CLIENT OR SERVER)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$sockaddr_to_name(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        unsigned char name[256],
        unsigned long *nlength,
        unsigned long *port,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$sockaddr_to_name(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out name: array [0..255] of char;
        in out nlength: unsigned32;
        out port: unsigned32;
        out status: status_$t);
```

DESCRIPTION

This call is obsolete. See the note below.

The **rpc_$sockaddr_to_name** call provides the address family, the host name, and the port number identified by the specified socket address.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

name A string that contains the host name and the address family. The format is *family:host*, where *family* can be either dds or ip.

nlength On input, *nlength* is the length of the *name* buffer. On output, *nlength* is the number of characters returned in the *name* parameter.

port The socket port number.

status The completion status.

NOTE

This call has been superseded by the **socket_$to_name** call.

FILES

idl/rpc.idl

SEE ALSO

rpc_$name_to_sockaddr, socket_$to_name

NAME

rpc_$unregister – unregister an interface (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$unregister(
        rpc_$if_spec_t *ifspec,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$unregister(
        in ifspec: rpc_$if_spec_t;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$unregister** call unregisters an interface that the server previously registered with the RPC runtime library. After an interface is unregistered, the RPC runtime library will not pass requests for that interface to the server.

If a server uses several **rpc_$register** or **rpc_$register_mgr** calls to register an interface more that once, then it must call **rpc_$unregister** an equal number of times to unregister the interface.

ifspec An interface specifier. The interface being unregistered.

status The completion status.

EXAMPLE

The following statement from *examples*/**stacks/server.c**, the **stacks** example server program, unregisters the **stacks** interface from the RPC runtime library:

```
rpc_$unregister (&stacks_v1$if_spec, &stat);
```

FILES

idl/**rpc.idl**

SEE ALSO

rpc_$register, rpc_$register_mgr, rpc_$register_object

NAME

rpc_$use_family – create a socket of a specified address family for an RPC server (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$use_family(
        unsigned long family,
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$use_family(
        in family: unsigned32;
        out sockaddr: socket_$addr_t;
        out slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$use_family** call creates a socket for a server without specifying its port number. The RPC runtime library assigns an opaque port number. If a server must listen on a particular well-known port, use **rpc_$use_family_wk** to create the socket.

A server can listen on more than one socket. However, a server ordinarily listens on only one socket per address family, regardless of how many interfaces it exports. Therefore, most servers should make this call once per address family.

family The address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the socket address is expressed. The **rpc_$intro** section describes possible values.

sockaddr The socket address of the socket on which the server will listen.

slength The length, in bytes, of *sockaddr*.

status The completion status.

ERRORS

rpc_$cant_create_sock
The RPC runtime library could not create a socket.

rpc_$cant_bind_sock
The RPC runtime library created a socket but could not bind it to a socket address.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called **rpc_$use_family** or **rpc_$use_family_wk** too many times.

EXAMPLE

The following statement from *examples*/**binoplu**/**server.c**, the **binoplu** example server program, creates a socket for the server:

```
rpc_$use_family (family, &loc, &llen, &st);
```

FILES

***idl*/rpc.idl**

SEE ALSO

rpc_$use_family_wk

NAME

rpc_$use_family_wk – create a socket with a well-known port for an RPC server (SERVER ONLY)

SYNOPSIS (C)

```
#include <idl/c/rpc.h>

void rpc_$use_family_wk(
        unsigned long family,
        rpc_$if_spec_t *ifspec,
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rpc.ins.pas'

procedure rpc_$use_family_wk(
        in family: unsigned32;
        in ifspec: rpc_$if_spec_t;
        out sockaddr: socket_$addr_t;
        out slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **rpc_$use_family_wk** call creates a socket that uses the port specified via the *ifspec* parameter. Use this call to create a socket only if a server must listen on a particular well-known port. Otherwise, use **rpc_$use_family**.

Most servers that use well-known ports should make this call once per address family.

family The address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the socket address is expressed. The **rpc_$intro** section describes possible values.

ifspec The interface that will be registered by the server. Typically, this parameter is the *interface*$if_spec generated by the NIDL Compiler from the interface definition; the well-known port is specified as an interface attribute.

sockaddr The socket address of the socket on which the server will listen.

slength The length, in bytes, of *sockaddr*.

status The completion status.

ERRORS

rpc_$cant_create_sock
The RPC runtime library could not create a socket.

rpc_$cant_bind_sock
The RPC runtime library created a socket but could not bind it to a socket address.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called **rpc_$use_family** or **rpc_$use_family_wk** too many times.

rpc_$use_family_wk

rpc_$addr_in_use
The specified address and port are already in use. This is caused by multiple calls to **rpc_$use_family_wk** with the same well-known port.

EXAMPLE

The following statement from *examples/***binopwk/server.c**, the **binopwk** example server program, creates a socket for the server:

```
rpc_$use_family_wk (family, &binopwk_v1$if_spec, &loc, &llen, &st);
```

FILES

***idl/*rpc.idl**

SEE ALSO

rpc_$use_family

Chapter 14

rrpc_$ Calls

Contents

Annexe A.7 :

Remote RPC RunTime

NAME

rrpc_$intro – Remote Remote Procedure Call interface

SYNOPSIS (C)

```
#include <idl/c/rrpc.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/rrpc.ins.pas';
```

DESCRIPTION

The **rrpc_$** calls enable a client to request information about a server or to shut down a server.

The **rrpc_** interface is defined by the file *idl*/**rrpc.idl**, where the symbol *idl* denotes the system idl directory. On Apollo workstations and other UNIX systems, *idl* is /usr/include/idl.

If you are using the **rrpc_** interface on Apollo systems, see Appendix B for more information.

CONSTANTS

rrpc_$mod
: A module code indicating the Remote RPC module. See the description of the **status_$t** type.

The **rrpc_$sv** constants are indexes for elements in an **rrpc_$stat_vec_t** array. Each element is a 32-bit integer representing a statistic about a server. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in
: The number of calls processed by the server.

rrpc_$sv_rcvd
: The number of packets received by the server.

rrpc_$sv_sent
: The number of packets sent by the server.

rrpc_$sv_calls_out
: The number of calls made by the server.

rrpc_$sv_frag_resends
: The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
: The number of duplicate fragments received by the server.

DATA TYPES

The following data types are used in **rpc_$** calls:

handle_t An RPC handle.

rrpc_$interface_vec_t
: An array of **rpc_$if_spec_t**, RPC interface specifiers.

rrpc_$stat_vec_t
: An array of 32-bit integers, indexed by **rrpc_$sv** constants, representing statistics about a server.

status_$t A status code. Most of the NCS calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

STATUS CODES

rrpc_$shutdown_not_allowed
: Remote shutdown of the server is not allowed. Either the server does not ever allow remote shutdown or the server executed a check function that returned "false."

status_$ok
: The call was successful.

FILES

idl/**rrpc.idl**

rrpc_$are_you_there

NAME

rrpc_$are_you_there – check whether a server is answering requests

SYNOPSIS (C)

```
#include <idl/c/rrpc.h>

void rrpc_$are_you_there(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rrpc.ins.pas'

procedure rrpc_$are_you_there(
        in handle: handle_t;
        out status: status_$t);
```

DESCRIPTION

The rrpc_$are_you_there call checks whether a server is answering requests. If the server is answering requests, the completion status of this call is status_$ok.

handle An RPC handle.

status The completion status.

FILES

idl/rrpc.idl

rrpc_$inq_interfaces

NAME

rrpc_$inq_interfaces – obtain a list of the interfaces that a server exports

SYNOPSIS (C)

```
#include <idl/c/rrpc.h>

void rrpc_$inq_interfaces(
        handle_t handle,
        unsigned long max_ifs,
        rrpc_$interface_vec_t ifs,
        unsigned long *l_if,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rrpc.ins.pas'

procedure rrpc_$inq_interfaces(
        in handle: handle_t;
        in max_ifs: unsigned32;
        out ifs: univ rrpc_$interface_vec_t;
        out l_if: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The rrpc_$inq_interfaces call returns an array of RPC interface specifiers.

handle An RPC handle.

max_ifs The maximum number of elements in the array of interface specifiers.

ifs An array of rpc_$if_spec_t.

l_if The index of the last element in the returned array.

status The completion status.

FILES

idl/rrpc.idl

NAME

rrpc_$inq_stats – obtain statistics about a server

SYNOPSIS (C)

```
#include <idl/c/rrpc.h>

void rrpc_$inq_stats(
        handle_t handle,
        unsigned long max_stats,
        rrpc_$stat_vec_t stats,
        unsigned long *l_stat,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rrpc.ins.pas'

procedure rrpc_$inq_stats(
        in handle: handle_t;
        in max_stats: unsigned32;
        out stats: univ rrpc_$stat_vec_t;
        out l_stat: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The rrpc_$inq_stats call returns an array of integer statistics about a server.

handle An RPC handle.

max_stats The maximum number of elements in the array of statistics.

stats An array of 32-bit integers representing statistics about the server. A set of rrpc_$sv constants defines indexes for the elements in this array. The following list describes the statistic indexed by each rrpc_$sv constant:

rrpc_$sv_calls_in
The number of calls processed by the server.

rrpc_$sv_rcvd
The number of packets received by the server.

rrpc_$sv_sent
The number of packets sent by the server.

rrpc_$sv_calls_out
The number of calls made by the server.

rrpc_$sv_frag_resends
The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
The number of duplicate fragments received by the server.

l_stat The index of the last element in the returned array.

status The completion status.

FILES

idl/rrpc.idl

rrpc_$shutdown

:

NAME

rrpc_$shutdown – shut down a server

SYNOPSIS (C)

```
#include <idl/c/rrpc.h>

void rrpc_$shutdown(
        handle_t handle,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/rrpc.ins.pas'

procedure rrpc_$shutdown(
        in handle: handle_t;
        out status: status_$t);
```

DESCRIPTION

The **rrpc_$shutdown** call shuts down a server, if the server allows it. A server can use the **rpc_$allow_remote_shutdown** call to allow or disallow remote shutdown.

handle An RPC handle.

status The completion status.

FILES

idl/rrpc.idl

SEE ALSO

rpc_$allow_remote_shutdown, rpc_$shutdown

Chapter 15

socket_$ Calls

Contents

Annexe A.8 :

RPC Socket

NAME

socket_$intro – operations on socket addresses

SYNOPSIS (C)

```
#include <idl/c/socket.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/socket.ins.pas';
```

DESCRIPTION

The socket_$ calls manipulate socket addresses. Unlike the calls provided in some operating systems, the socket_$ calls operate on addresses of any protocol family.

The socket_ interface is defined by the file *idl*/socket.idl, where the symbol *idl* denotes the system idl directory. On Apollo workstations and other UNIX systems, *idl* is /usr/include/idl.

CONSTANTS

The following constants are used in socket_$ calls:

socket_$addr_module_code
A module code indicating the socket address module. See the description of the status_$t type.

The socket_$eq constants are flags indicating the fields to be compared in a socket_$equal call.

socket_$eq_hostid
Indicates that the host IDs are to be compared.

socket_$eq_netaddr
Indicates that the network addresses are to be compared.

socket_$eq_port
Indicates that the port numbers are to be compared.

socket_$eq_network
Indicates that the network IDs are to be compared.

The following 16-bit-integer constants are values for the socket_$addr_family_t type, used to specify the address family in a socket_$addr_t structure. Note that several of the rpc_$ and socket_$ calls use the 32-bit-integer equivalents of these values.

socket_$unspec
Address family is unspecified.

socket_$internet
Internet Protocols (IP).

socket_$dds
Domain protocols (DDS).

socket_$unspec_port
A port number indicating that no port is specified.

The following 16-bit-integer constant is a value for the socket_$wk_ports_t type, used to specify a well-known port. Note that several of the socket_$ calls use the 32-bit-integer equivalent of this value.

socket_$wk_fwd
The Local Location Broker forwarding port.

DATA TYPES

The following data types are used in socket_$ calls:

socket_$addr_family_t
An enumerated type for specifying an address family. The "CONSTANTS" section lists values for this type.

socket_$addr_list_t
An array of socket addresses in socket_$addr_t format.

socket_$addr_t
A structure that uniquely identifies a socket address. This structure consists of a socket_$addr_family_t specifying an address family and 14 bytes specifying a socket address.

socket_$host_id_t
A structure that uniquely identifies a host. This structure consists of a socket_$addr_family_t specifying an address family and 12 bytes specifying a host.

socket_$len_list_t
An array of unsigned 32-bit integers, the lengths of socket addresses in a socket_$addr_list_t.

socket_$local_sockaddr_t
An array of 50 characters, used to store a socket address in a format native to the local host.

socket_$net_addr_t
A structure that uniquely identifies a network address. This structure consists of a socket_$addr_family_t specifying an address family and 12 bytes specifying a network address. It contains both a host ID and a network ID.

socket_$string_t
An array of 100 characters, used to store the string representation of an address family or a socket address.

The string representation of an address family is a textual name such as dds, ip, or unspec.

The string representation of a socket address has the format *family*:*host*[*port*], where *family* is the textual name of an address family, *host* is either a textual host name or a numeric host ID preceded by a #, and *port* is a port number.

socket_$wk_ports_t
An enumerated type for specifying a well-known port. The "CONSTANTS" section lists values for this type.

socket_$intro

status_$t A status code. Most of the NCS calls supply their completion status in this format. The status_$t type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use status_$t as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If all is equal to status_$ok, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

STATUS CODES

The following are status codes returned by socket_$ calls:

socket_$bad_numeric_name
A specified name in numeric format is invalid.

socket_$buff_too_large
A specified buffer size (for example, the length of a name) is too large.

socket_$buff_too_small
A specified buffer size (for example, the length of a name) is too small.

socket_$cant_create_socket
A socket could not be created.

socket_$cant_cvrt_addr_to_name
A specified address could not be converted to a name.

socket_$cant_find_name
A specified name could not be resolved to an address.

socket_$cant_get_if_config
The interface configuration list for the local host could not be obtained.

socket_$intro

socket_$cant_get_local_name
The name of the local host could not be obtained.

socket_$family_not_valid
The specified address family is not valid for the local host.

socket_$internal_error
An internal error.

socket_$invalid_name_format
The format of a specified name is invalid.

status_$ok
The call was successful.

FILES

idl/socket.idl

NAME

socket_$equal – compare two socket addresses

SYNOPSIS (C)

```
#include <idl/c/socket.h>

boolean socket_$equal(
        socket_$addr_t *sockaddr1,
        unsigned long s1length,
        socket_$addr_t *sockaddr2,
        unsigned long s2length,
        unsigned long flags,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

function socket_$equal(
        in sockaddr1: socket_$addr_t;
        in s1length: unsigned32;
        in sockaddr2: socket_$addr_t;
        in s2length: unsigned32;
        in flags: unsigned32;
        out status: status_$t): boolean;
```

DESCRIPTION

The **socket_$equal** call compares two socket addresses. The *flags* parameter determines which fields of the socket addresses are compared. The call returns "true" (nonzero) if all of the fields compared are equal, "false" (zero) if not.

sockaddr1
: A socket address.

s1length The length, in bytes, of *sockaddr1*.

sockaddr2
: A socket address.

s2length The length, in bytes, of *sockaddr2*.

flags The logical OR of values selected from the following:

socket_$eq_hostid
: Indicates that the host IDs are to be compared.

socket_$eq_netaddr
: Indicates that the network addresses are to be compared.

socket_$eq_port
: Indicates that the port numbers are to be compared.

socket_$eq_network
: Indicates that the network IDs are to be compared.

status The completion status.

EXAMPLE

The following call compares the network and host IDs in the socket addresses **sockaddr1** and **sockaddr2**:

```
if (socket_$equal (&sockaddr1, s1length, &sockaddr2, s2length,
        socket_$eq_network | socket_$eq_hostid, &st))
    printf ("sockaddrs have equal network and host IDs\n");
```

FILES

idl/socket.idl

NAME

socket_$family_from_name – convert an address family name to an integer

SYNOPSIS (C)

```
#include <idl/c/socket.h>

unsigned long socket_$family_from_name(
        socket_$string_t name,
        unsigned long nlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

function socket_$family_from_name(
        in name: socket_$string_t;
        in nlength: unsigned32;
        out status: status_$t): unsigned32;
```

DESCRIPTION

The **socket_$family_from_name** call returns the integer representation of the address family specified in the text string *name*.

name The textual name of an address family. Possible values include dds and ip.

nlength The length, in bytes, of *name*.

status The completion status.

EXAMPLE

The server program for the **binoplu** example, *examples*/**binoplu/server.c**, accepts a textual family name as its first argument. The program uses the following **socket_$family_from_name** call to convert this name to the corresponding integer representation:

```
family = socket_$family_from_name ((ndr_$char *)argv[1],
    (long)strlen(argv[1]), &st);
```

FILES

idl/**socket.idl**

SEE ALSO

socket_$family_to_name, socket_$from_name, socket_$to_name

NAME

socket_$family_to_name – convert an integer address family to a textual name

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$family_to_name(
        unsigned long family,
        socket_$string_t name,
        unsigned long *nlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$family_to_name(
        in family: unsigned32;
        out name: socket_$string_t;
        in out nlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$family_to_name** call converts the integer representation of an address family to a textual name for the family.

family The integer representation of an address family.

name The textual name of *family*.

nlength On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the returned name.

status The completion status.

EXAMPLE

The following statement converts the integer representation of an address family (**i**) to the corresponding textual name (**name**):

```
socket_$family_to_name (i, (ndr_$char *)name, &namelen, &st);
```

FILES

idl/**socket.idl**

SEE ALSO

socket_$family_from_name, socket_$from_name, socket_$to_name

NAME

socket_$from_local_rep – convert from a local representation of a socket address to a socket_$addr_t

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$from_local_rep(
        socket_$addr_t *sockaddr,
        socket_$local_sockaddr_t local_sockaddr,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$from_local_rep(
        in out sockaddr: socket_$addr_t;
        in local_sockaddr: socket_$local_sockaddr_t;
        out status: status_$t);
```

DESCRIPTION

The socket_$from_local_rep call converts a socket address from the format native to the local host into NCS socket_$addr_t format. This call is useful only on systems with non-standard socket address structure layouts, and even then, only if NCS-based applications need to use the native socket primitives on NCS socket_$addr_t structures.

sockaddr The representation of *local_sockaddr* in NCS socket_$addr_t format.

local_sockaddr
A socket address in the format native to the local host.

status The completion status.

FILES

idl/socket.idl

SEE ALSO

socket_$to_local_rep

NAME

socket_$from_name – convert a name and port number to a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$from_name(
        unsigned long family,
        socket_$string_t name,
        unsigned long nlength,
        unsigned long port,
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$from_name(
        in family: unsigned32;
        in name: socket_$string_t;
        in nlength: unsigned32;
        in port: unsigned32;
        out sockaddr: socket_$addr_t;
        in out slength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The socket_$from_name call converts a textual address family, host name, and port number to a socket address. The address family and the port number can be either specified as separate parameters or included in the *name* parameter.

family The integer representation of an address family. If the *family* parameter is socket_$unspec, then the *name* parameter is scanned for a prefix of *family*: (for example, ip:).

name A string in the format *family*:*host*[*port*], where *family*:, *host*, and [*port*] are all optional.

The *family* is an address family. Possible values include dds and ip. If you specify a *family* as part of the *name* parameter, you must specify socket_$unspec in the *family* parameter.

The *host* is a host name. A leading # can be used to indicate that the host name is in the standard numeric form (for example, #192.9.8.7 or #464a.465c). If *host* is omitted, the local host name is used.

The *port* is a port number. If you specify a *port* as part of the *name* parameter, the *port* parameter is ignored.

nlength The length, in bytes, of *name*.

port A port number. If you specify a port number in the *name* parameter, this parameter is ignored.

sockaddr A socket address.

socket_$from_name

slength The length, in bytes, of *sockaddr*.

status The completion status.

EXAMPLE

The client program for the **binopfw** example, *examples*/**binopfw/client.c**, accepts as its first argument a string identifying a server host, in the format *family*:*host*. The client program does not require the server port number because it sends its first remote procedure call to the Local Location Broker forwarding port at the specified host. The program uses the following **socket_$from_name** call to convert this string to a socket address:

```
socket_$from_name ((long)socket_$unspec, (ndr_$char *)argv[1],
    (long)strlen(argv[1]), (long)socket_$unspec_port, &loc, &llen, &st);
```

FILES

***idl*/socket.idl**

SEE ALSO

socket_$family_from_name, socket_$to_name

socket_$inq_broad_addrs

NAME

socket_$inq_broad_addrs – return a list of broadcast addresses

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$inq_broad_addrs(
        unsigned long family,
        unsigned long port,
        socket_$addr_list_t brd_addrs,
        socket_$len_list_t brd_lens,
        unsigned long *length,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$inq_broad_addrs(
        in family: unsigned32;
        in port: unsigned32;
        out brd_addrs: univ socket_$addr_list_t;
        out brd_lens: univ socket_$len_list_t;
        in out length: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$inq_broad_addrs** call returns a list of all broadcast addresses that the local host can use in the specified address family.

If a host has network interfaces to several networks in *family*, this call returns a socket address for each interface that supports broadcasting. IP interfaces via serial lines, for example, do not support broadcasting, so no addresses are returned for such interfaces.

family The integer representation of an address family.

port The value to be used as the port number in the returned addresses.

brd_addrs
An array of the socket addresses in *family* to which the host can send broadcasts.

brd_lens An array of the lengths of the *brd_addrs*.

length On input, the maximum number of addresses to be returned in *brd_addrs*. On return, the number of addresses actually returned.

status The completion status.

FILES

***idl*/socket.idl**

NAME

socket_$inq_hostid – return the host ID part of a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$inq_hostid(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        socket_$host_id_t *hostid,
        unsigned long *hlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$inq_hostid(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out hostid: socket_$host_id_t;
        in out hlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$inq_hostid** call returns the host ID part of a socket address.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

hostid The host ID part of *sockaddr*.

hlength The length, in bytes, of *hostid*.

status The completion status.

FILES

idl/**socket.idl**

SEE ALSO

socket_$set_hostid

NAME

socket_$inq_my_netaddr – return the primary network address for the local host

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$inq_my_netaddr(
        unsigned long family,
        socket_$net_addr_t *netaddr,
        unsigned long *nlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$inq_my_netaddr(
        in family: unsigned32;
        out netaddr: socket_$net_addr_t;
        in out nlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$inq_my_netaddr** call returns the primary network address for the local host in the specified family.

family The integer representation of an address family.

netaddr The network address for the local host in *family*.

nlength The length, in bytes, of *netaddr*.

status The completion status.

FILES

idl/**socket.idl**

SEE ALSO

socket_$inq_netaddr, socket_$set_netaddr

NAME

socket_$inq_netaddr – return the network address part of a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$inq_netaddr(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        socket_$net_addr_t *netaddr,
        unsigned long *nlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$inq_netaddr(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out netaddr: socket_$net_addr_t;
        in out nlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The socket_$inq_netaddr call returns the network address part of a socket address.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

netaddr The network address part of *sockaddr*.

nlength The length, in bytes, of *netaddr*.

status The completion status.

FILES

idl/socket.idl

SEE ALSO

socket_$inq_my_netaddr, socket_$set_netaddr

NAME

socket_$inq_port – return the port number part of a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

unsigned long socket_$inq_port(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

function socket_$inq_port(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out status: status_$t): unsigned32;
```

DESCRIPTION

The socket_$inq_port call returns the port number part of a socket address.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

status The completion status.

EXAMPLE

The following call returns the port number in the socket address sockaddr:

```
port = socket_$inq_port (&sockaddr, slen, &st);
```

FILES

idl/socket.idl

SEE ALSO

socket_$inq_my_netaddr, socket_$inq_netaddr, socket_$set_port

socket_$max_pkt_size

NAME

socket_$max_pkt_size – return the maximum packet size for an address family

SYNOPSIS (C)

```
#include <idl/c/socket.h>

unsigned long socket_$max_pkt_size(
        unsigned long family,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

function socket_$max_pkt_size(
        in family: unsigned32;
        out status: status_$t): unsigned32;
```

DESCRIPTION

The **socket_$max_pkt_size** call returns the maximum packet size, in bytes, for the specified address family.

family The integer representation of an address family. Possible values include **socket_$internet** and **socket_$dds**.

status The completion status.

FILES

idl/**socket.idl**

socket_$set_hostid

NAME

socket_$set_hostid – set the host ID part of a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$set_hostid(
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        socket_$host_id_t *hostid,
        unsigned long hlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$set_hostid(
        in out sockaddr: socket_$addr_t;
        in out slength: unsigned32;
        in hostid: socket_$host_id_t;
        in hlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$set_hostid** call sets the host ID in a socket address to the specified value.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

hostid A host ID.

hlength The length, in bytes, of *hostid*.

status The completion status.

FILES

idl/**socket.idl**

SEE ALSO

socket_$inq_hostid

NAME

socket_$set_netaddr – set the network address part of a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$set_netaddr(
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        socket_$net_addr_t *netaddr,
        unsigned long nlength,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$set_netaddr(
        in out sockaddr: socket_$addr_t;
        in out slength: unsigned32;
        in netaddr: socket_$net_addr_t;
        in nlength: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$set_netaddr** call sets the network address in a socket address to the specified value.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

netaddr A network address.

nlength The length, in bytes, of *netaddr*.

status The completion status.

SEE ALSO

socket_$inq_netaddr

NAME

socket_$set_port – set the port number in a socket address

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$set_port(
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        unsigned long port,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$set_port(
        in out sockaddr: socket_$addr_t;
        in out slength: unsigned32;
        in port: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$set_port** call sets the port number in a socket address to the specified value.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

port The value to which the port number in *sockaddr* will be set.

status The completion status.

EXAMPLE

The following call sets the port number in **sockaddr** to **port**:

```
socket_$set_port (&sockaddr, &slen, port, &st);
```

SEE ALSO

socket_$inq_port, socket_$set_netaddr, socket_$set_wk_port

NAME

socket_$set_wk_port – set the port number in a socket address to a well-known value

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$set_wk_port(
        socket_$addr_t *sockaddr,
        unsigned long *slength,
        unsigned long port,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$set_wk_port(
        in out sockaddr: socket_$addr_t;
        in out slength: unsigned32;
        in port: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$set_wk_port** call sets the port number in a socket address to the specified well-known value.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

port A value of the enumerated type **socket_$wk_ports_t**. The well-known value to which the port number in *sockaddr* will be set.

status The completion status.

EXAMPLE

The Local Location Broker daemon listens on the LLB forwarding port, which has the well-known port number **socket_$wk_fwd**. The daemon uses the following call to set the port number in its socket address:

```
socket_$set_wk_port (&saddr, &...  (unsigned long)socket_$wk_fwd, &st);
```

SEE ALSO

socket_$inq_port, socket_$set_netaddr, socket_$set_port

NAME

socket_$to_local_rep – convert a **socket_$addr_t** to a local representation

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$to_local_rep(
        socket_$addr_t *sockaddr,
        socket_$local_sockaddr_t local_sockaddr,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$to_local_rep(
        in sockaddr: socket_$addr_t;
        in out local_sockaddr: socket_$local_sockaddr_t;
        out status: status_$t);
```

DESCRIPTION

The **socket_$to_local_rep** call converts a socket address from NCS **socket_$addr_t** format into a format usable by the socket primitives native to the local host. This call is useful only on systems with non-standard socket address structure layouts, and even then, only if NCS-based applications need to use the native socket primitives on NCS **socket_$addr_t** structures.

sockaddr A socket address in NCS **socket_$addr_t** format.

local_sockaddr
The representation of *sockaddr* in a format native to the local host.

status The completion status.

FILES

idl/**socket.idl**

SEE ALSO

socket_$from_local_rep

NAME

socket_$to_name – convert a socket address to a name and port number

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$to_name(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        socket_$string_t name,
        unsigned long *nlength,
        unsigned long *port,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$to_name(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out name: socket_$string_t;
        in out nlength: unsigned32;
        out port: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$to_name** call converts a socket address to a textual address family, host name, and port number.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

name A string in the format *family*:*host*, where *family* is the address family and *host* is the host name; *host* may be in the standard numeric form (for example, #192.1.2.3 or #499d.488a) if a textual host name cannot be obtained.

nlength On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the name returned.

port The port number.

status The completion status.

EXAMPLE

The client program for the **binoplu** example, *examples*/**binoplu/client.c**, uses the following call to convert the socket address for its server to a textual name:

```
socket_$to_name (&loc, llen, name, &namelen, &port, &st);
```


FILES

idl/socket.idl

SEE ALSO

socket_$family_to_name, socket_$from_name, socket_$to_numeric_name

socket_$to_numeric_name

NAME

socket_$to_numeric_name – convert a socket address to a numeric name and port number

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$to_numeric_name(
        socket_$addr_t *sockaddr,
        unsigned long slength,
        socket_$string_t name,
        unsigned long *nlength,
        unsigned long *port,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$to_numeric_name(
        in sockaddr: socket_$addr_t;
        in slength: unsigned32;
        out name: socket_$string_t;
        in out nlength: unsigned32;
        out port: unsigned32;
        out status: status_$t);
```

DESCRIPTION

The **socket_$to_numeric_name** call converts a socket address to a textual address family, a numeric host name, and a port number.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

name A string in the format *family:host*, where *family* is the address family and *host* is the host name in the standard numeric form (for example, #192.7.8.9 for an IP address or #840c.940f for a DDS address).

nlength On input, the maximum length, in bytes, of the numeric name to be returned. On output, the actual length of the name returned.

port The port number.

status The completion status.

FILES

idl/socket.idl

SEE ALSO

socket_$family_to_name, socket_$from_name, socket_$to_name

socket_$valid_families

NAME

socket_$valid_families – obtain a list of valid address families

SYNOPSIS (C)

```
#include <idl/c/socket.h>

void socket_$valid_families(
        unsigned long *max_families,
        socket_$addr_family_t families[],
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

procedure socket_$valid_families(
        in out max_families: unsigned32;
        out families: array [1..*] of socket_$addr_family_t;
        out status: status_$t);
```

DESCRIPTION

The **socket_$valid_families** call returns a list of the address families that are valid for the NCK implementation on the calling host.

max_families
The maximum number of families that can be returned.

families An array of **socket_$addr_family_t**. Possible values for this type are enumerated in *idl*/nbase.idl.

status The completion status. A status of **socket_$buff_too_small** indicates that the *families* array is not long enough to hold all the valid families.

EXAMPLE

The following call returns a list of at most two valid address families:

```
socket_$valid_families (2, &families, $st);
```

FILES

idl/socket.idl

SEE ALSO

socket_$valid_family

NAME

socket_$valid_family – check whether an address family is valid

SYNOPSIS (C)

```
#include <idl/c/socket.h>

boolean socket_$valid_family(
        unsigned long family,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/socket.ins.pas'

function socket_$valid_family(
        in family: unsigned32;
        out status: status_$t): boolean;
```

DESCRIPTION

The socket_$valid_family call returns "true" if the specified address family is valid for the NCK implementation on the calling host, "false" if not.

family The integer representation of an address family, as enumerated in *idl*/nbase.idl.

status The completion status.

EXAMPLE

The following call checks whether socket_$internet is a valid address family:

```
internetvalid = socket_$valid_family (socket_$internet, &st);
```

FILES

idl/socket.idl

SEE ALSO

socket_$valid_families

Chapter 16

uuid_$ Calls

Contents

Annexe A.9 :

RPC UUID

uuid_$intro

NAME

uuid_$intro – operations on Universal Unique Identifiers

SYNOPSIS (C)

```
#include <idl/c/uuid.h>
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/uuid.ins.pas';
```

DESCRIPTION

The **uuid_$** calls operate on UUIDs (Universal Unique Identifiers).

The **uuid_** interface is defined by the file *idl*/**uuid.idl**, where the symbol *idl* denotes the system **idl** directory. On Apollo workstations and other UNIX systems, *idl* is /usr/include/idl.

Apollo software also uses UIDs (Unique Identifiers) as identifiers. On Apollo systems, we supply two additional operations that convert UUIDs to UIDs and vice versa, and we define these operations in the file *idl*/**uuid_uid.idl**.

EXTERNAL VARIABLES

The following external variables are used in **uuid_$** calls:

uid_$nil An external **uid_$t** variable that is preassigned the value of the nil UID. Do not change the value of this variable.

uuid_$nil
An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

DATA TYPES

The following data types are used in **uuid_$** calls.

status_$t A status code. Most of the NCS calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```


uuid_$intro

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uid_$t A 64-bit value that uniquely identifies objects and types on Apollo systems.

uuid_$string_t
A string of 37 characters (including a null terminator) that is an ASCII representation of a UUID. The format is *cccccccccccc.ff.h1.h2.h3.h4.h5.h6.h7*, where *cccccccccccc* is the timestamp, *ff* is the address family, and *h1* ... *h7* are the seven bytes of host identifier. Each character in these fields is a hexadecimal digit.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. The **uuid_$t** type is defined as follows:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

time_high
The high 32 bits of a 48-bit unsigned time value which is the number of 4-microsecond intervals that have passed between 1 January 1980 00:00 GMT and the time of UUID creation.

time_low The low 16 bits of the 48-bit time value.

reserved 16 bits of reserved space.

family 8 bits identifying an address family.

host 7 bytes identifying the host on which the UUID was created. The format of this field depends on the address family.

STATUS CODES

status_$ok
The call was successful.

FILES

idl/**uuid.idl**

NAME

uuid_$decode – convert a character-string representation of a UUID into a UUID

SYNOPSIS (C)

```
#include <idl/c/uuid.h>

void uuid_$decode(
        uuid_$string_t s,
        uuid_$t *uuid,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid.ins.pas'

procedure uuid_$decode(
        in s: uuid_$string_t;
        out uuid: uuid_$t;
        out status: status_$t);
```

DESCRIPTION

The **uuid_$decode** call returns the UUID corresponding to a valid character-string representation of a UUID.

s The character-string representation of a UUID.

uuid The UUID that corresponds to *s*.

status A **status_$t**. The completion status.

EXAMPLE

The following call returns as **leek_uuid** the UUID corresponding to the character-string representation in **leek_uuid_rep**:

```
uuid_$decode (leek_uuid_rep, &leek_uuid, &status);
```

FILES

***idl*/uuid.idl**

SEE ALSO

uuid_$encode

NAME

uuid_$encode – convert a UUID into its character-string representation

SYNOPSIS (C)

```
#include <idl/c/uuid.h>

void uuid_$encode(
        uuid_$t *uuid,
        uuid_$string_t s)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid.ins.pas'

procedure uuid_$encode(
        in uuid: uuid_$t;
        out s: uuid_$string_t);
```

DESCRIPTION

The **uuid_$encode** call returns the character-string representation of a UUID.

uuid A UUID.

s The character-string representation of *uuid*.

EXAMPLE

The following call returns as **shallot_uuid_rep** the character-string representation for the UUID **shallot_uuid**:

```
uuid_$encode (&shallot_uuid, shallot_uuid_rep);
```

FILES

***idl*/uuid.idl**

SEE ALSO

uuid_$decode

uuid_$equal

NAME

uuid_$equal – compare two UUIDs

SYNOPSIS (C)

```
#include <idl/c/uuid.h>

boolean uuid_$equal(
        uuid_$t *u1,
        uuid_$t *u2)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid.ins.pas'

function uuid_$equal(
        in u1: uuid_$t;
        in u2: uuid_$t): boolean;
```

DESCRIPTION

The **uuid_$equal** call compares the UUIDs *u1* and *u2*. It returns "true" if they are equal, "false" if they are not.

u1 A UUID.

u2 Another UUID.

EXAMPLE

The following code compares the UUIDs **cilantro_uuid** and **coriander_uuid**:

```
if (uuid_$equal (&cilantro_uuid, &coriander_uuid))
    printf ("cilantro and coriander UUIDs are equal\n");
else
    printf ("cilantro and coriander UUIDs are not equal\n");
```

FILES

idl/**uuid.idl**

uuid_$from_uid

NAME

uuid_$from_uid – convert a Domain UID into a UUID (APOLLO SYSTEMS ONLY)

SYNOPSIS (C)

```
#include <idl/c/uuid_uid.h>

void uuid_$from_uid(
        uid_$t *uid,
        uuid_$t *uuid)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid_uid.ins.pas'

procedure uuid_$from_uid(
        in uid: uid_$t;
        out uuid: uuid_$t);
```

DESCRIPTION

The **uuid_$from_uid** call returns the 128-bit NCS UUID corresponding to a 64-bit Domain UID. This call is available only on Apollo systems.

uid A Domain Unique Identifier (UID).

uuid The UUID corresponding to *uid*.

EXAMPLE

The following call returns as **onion_uuid** the UUID corresponding to the UID **onion_uid**:

```
uuid_$from_uid (&onion_uid, &onion_uuid);
```

FILES

idl/**uuid.idl**

SEE ALSO

uuid_$to_uid

uuid_$gen

NAME

uuid_$gen – generate a new UUID

SYNOPSIS (C)

```
#include <idl/c/uuid.h>

void uuid_$gen(
        uuid_$t *uuid)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid.ins.pas'

procedure uuid_$gen(
        out uuid: uuid_$t);
```

DESCRIPTION

The **uuid_$gen** call returns a new UUID.

uuid A UUID.

EXAMPLE

The following call returns as **newid** a new UUID:

```
uuid_$gen (&newid);
```

FILES

idl/**uuid.idl**

uuid_$to_uic

NAME

uuid_$to_uid – convert a UUID into a Domain UID (APOLLO SYSTEMS ONLY)

SYNOPSIS (C)

```
#include <idl/c/uuid_uid.h>

void uuid_$to_uid(
        uuid_$t *uuid,
        uid_$t *uid,
        status_$t *status)
```

SYNOPSIS (PASCAL)

```
%include '/usr/include/idl/pas/uuid_uid.ins.pas'

procedure uuid_$to_uid(
        in uuid: uuid_$t;
        out uid: uid_$t;
        out status: status_$t);
```

DESCRIPTION

The **uuid_$to_uid** call returns the 64-bit Domain UID corresponding to a 128-bit NCS UUID. This call is available only on Apollo systems. It is not guaranteed to work, since not all UUIDs correspond to UIDs.

uuid A UUID.

uid The Domain Unique Identifier (UID) corresponding to *uuid*.

status A **status_$t**. The completion status.

EXAMPLE

The following call returns as **scallion_uid** the UID corresponding to the UUID **scallion_uuid**:

```
uuid_$to_uid (&scallion_uuid, &scallion_uid, &st);
```

FILES

idl/**uuid.idl**

SEE ALSO

uuid_$from_uid

Annexe B:

Messages d'erreur du NCS

Appendix C

NCA Status Codes

This appendix shows the interface definition file **ncastat.idl**, which describes the remote interface **nca_status_**. The **nca_status_** remote interface consists of constant definitions for the NCA-defined status codes, in NIDL/Pascal syntax. NCA status codes are "well-known" and are part of the NCA/RPC protocol; that is, they are returned in *fault* or *reject* messages from servers. The first few status codes are derived from status codes that the Network Computing System (NCS) uses (NCS is Apollo's implementation of NCA.) Table 4-6 gives descriptions of these status codes.

```
%pascal

[uuid(3c667ff91000.0d.00.01.34.22.00.00.00)] interface nca_status_;

const
    nca_status_$comm_failure        = 16#1C010001;
    nca_status_$op_rng_error        = 16#1C010002;
    nca_status_$unk_if              = 16#1C010003;
    nca_status_$wrong_boot_time     = 16#1C010006;
    nca_status_$you_crashed         = 16#1C010009;
    nca_status_$proto_error         = 16#1C01000B;
    nca_status_$out_args_too_big    = 16#1C010013;
    nca_status_$server_too_busy     = 16#1C010014;
    nca_status_$unsupported_type    = 16#1C010017;
    nca_status_$zero_divide         = 16#1C000001;
    nca_status_$address_error       = 16#1C000002;
    nca_status_$fp_div_zero         = 16#1C000003;
    nca_status_$fp_underflow        = 16#1C000004;
    nca_status_$fp_overflow         = 16#1C000005;
    nca_status_$invalid_tag         = 16#1C000006;
    nca_status_$invalid_bound       = 16#1C000007;
end;
```

Annexe C:

Syntaxe NIDL (fichier YACC)

Appendix A

NIDL yacc Input Specification

The yacc input file for NIDL consists of a declaration section that defines the NIDL tokens (a token is an entity that cannot be further decomposed, such as a number or an identifier) and a rules section that gives the grammar rules for NIDL/Pascal and NIDL/C. Each yacc rule references a token or another yacc rule. Representing the NIDL grammar as a yacc input file produces an unambiguous specification of NIDL/Pascal and NIDL/C and allows the specification to be used as input to yacc to generate NIDL/C and NIDL/Pascal parsers. The NIDL specification given in this appendix follows the conventions and notation for yacc input file generation. See the 4.3BSD *UNIX Programmer's Supplementary Documents, Volume 1* for a complete description of these conventions.

The yacc input on the pages that follow gives NIDL keywords, common NIDL syntax, NIDL/Pascal–specific syntax, and NIDL/C–specific syntax, respectively. Note that the printed representations of keywords are case insensitive in NIDL/Pascal and case sensitive in NIDL/C.

```
%{

%token ARRAY_KW
%token BITSET_KW
%token BOOLEAN_KW
%token BYTE_KW
%token CHAR_KW
%token CASE_KW
%token COMM_STATUS_KW
%token CONST_KW
%token DOUBLE_KW
%token END_KW
%token FROM_KW
%token FUNCTION_KW
%token IDEMPOTENT_KW
%token IN_KW
```

```
%token IMPLICIT_HANDLE_KW
%token IMPORT_KW
%token INCLUDE_KW
%token INTEGER_KW
%token INTEGER8_KW
%token INTEGER32_KW
%token INTEGER64_KW
%token INTERFACE_KW
%token LAST_IS_KW
%token MAYBE_KW
%token MAX_IS_KW
%token NIL_KW
%token OF_KW
%token OTHERWISE_KW
%token OUT_KW
%token PORT_KW
%token PROCEDURE_KW
%token RECORD_KW
%token REAL_KW
%token REMOTE_KW
%token SET_KW
%token STRINGO_KW
%token TAG_IS_KW
%token TYPE_KW
%token UUID_KW
%token UNION_KW
%token UNSIGNED_KW
%token UNSIGNED8_KW
%token UNSIGNED32_KW
%token UNSIGNED64_KW
%token VERSION_KW
%token HYPER_KW
%token LONG_KW
%token SHORT_KW
%token FLOAT_KW
%token VOID_KW
%token SMALL_KW
%token SWITCH_KW
%token TYPEDEF_KW
%token STRUCT_KW
%token ENUM_KW
%token INT_KW
%token REF_KW
%token OPTION_KW
%token HANDLE_T_KW
%token HANDLE_KW
%token UUID_REP
%token TRANSMIT_AS_KW
%token TRUE_KW
%token FALSE_KW
%token BROADCAST_KW
%token UNIV_PTR_KW
```



```
%token COLON
%token COMMA
%token DOTDOT
%token EQUAL
%token HAT
%token LBRACE
%token LBRACKET
%token LPAREN
%token RBRACE
%token RBRACKET
%token RPAREN
%token SEMI
%token STAR
%token AMPER
%token IDENTIFIER
%token STRING
%token INTEGER_NUMERIC

%start interface
%%
interface:
        interface_attributes INTERFACE_KW IDENTIFIER interface_tail
    ;
interface_tail:
        pascal_interface_tail
    |   c_interface_tail
    ;
pascal_interface_tail:
        pas_interface_marker interface_body pas_interface_close
    ;
pas_interface_marker:
        SEMI
    ;
pas_interface_close:
        END_KW SEMI
    ;
interface_attributes:
        attribute_opener interface_attr_list attribute_closer
    |   /* Nothing */
    ;
attribute_opener:
        LBRACKET
    ;
attribute_closer:
        RBRACKET
    ;

interface_attr_list:
        interface_attr
    |   interface_attr_list COMMA interface_attr
    ;
```

```
interface_attr:
        IMPLICIT_HANDLE_KW LPAREN IDENTIFIER COLON
            builtin_type_exp RPAREN
    |   IMPLICIT_HANDLE_KW LPAREN c_simple_type_spec IDENTIFIER  RPAREN
    |   UUID_KW UUID_REP
    |   PORT_KW LPAREN port_list RPAREN
    |   VERSION_KW LPAREN INTEGER_NUMERIC RPAREN
    ;

port_list:
        port_spec
    |   port_list COMMA port_spec
    ;
port_spec:
        IDENTIFIER COLON LBRACKET INTEGER_NUMERIC RBRACKET
    ;
interface_body:
        exports
    |   imports exports
    |   /* nothing */
    ;
imports:
        import
    |   imports import
    ;
import:
        IMPORT_KW import_list SEMI
    ;
import_list:
        import
    |   import_list COMMA import
    ;
import:
        STRING
    ;
exports:
        export
    |   exports export
    ;

export:
        CONST_KW const_defs
    |   TYPE_KW  type_defs
    |   proc_def
    |   func_def
    ;

const_defs:
        const_def
    |   const_defs const_def
    ;
const_def:
```



```
        IDENTIFIER EQUAL const_exp  SEMI
    ;
const_exp:
        INTEGER_NUMERIC
    |   IDENTIFIER
    |   STRING
    |   NIL_KW
    |   TRUE_KW
    |   FALSE_KW
    ;

type_defs:
        type_def
    |   type_defs type_def
    ;
type_def:
        IDENTIFIER EQUAL attributed_type SEMI
    ;
attributed_type:
        type_attribute_list  type_exp
    |   type_exp
    ;
type_attribute_list:
        attribute_opener type_attributes attribute_closer
    ;
type_attributes:
        type_attribute
    |   type_attributes COMMA type_attribute
    ;
type_attribute:
        LAST_IS_KW LPAREN IDENTIFIER RPAREN
    |   MAX_IS_KW LPAREN IDENTIFIER RPAREN
    |   HANDLE_KW
    |   TRANSMIT_AS_KW LPAREN IDENTIFIER RPAREN
    ;
type_exp:
        simple_type_exp
    |   structured_type_exp
    ;
simple_type_exp:
        builtin_type_exp
    |   enumerated_type_exp
    |   subrange_type_exp
    ;


builtin_type_exp:
        BOOLEAN_KW
    |   BYTE_KW
    |   CHAR_KW
    |   INTEGER_KW
    |   INTEGER8_KW
```

```
    |   INTEGER32_KW
    |   INTEGER64_KW
    |   UNSIGNED_KW
    |   UNSIGNED8_KW
    |   UNSIGNED32_KW
    |   UNSIGNED64_KW
    |   REAL_KW
    |   DOUBLE_KW
    |   HANDLE_T_KW
    |   IDENTIFIER
    |   UNIV_PTR_KW
    ;
enumerated_type_exp:
    |   LPAREN enum_ids RPAREN
    ;
enum_ids:
        enum_id
    |   enum_ids COMMA enum_id
    ;
enum_id:
        IDENTIFIER
    ;
subrange_type_exp:
        const_exp DOTDOT const_exp
    ;
structured_type_exp:
        open_array_type_exp
    |   fixed_array_type_exp
    |   ptr_type_exp
    |   record_type_exp
    |   set_type_exp
    |   proc_ptr_type_exp
    |   func_ptr_type_exp
    |   string0_type_exp
    ;

open_array_type_exp:
        ARRAY_KW LBRACKET open_array_index RBRACKET OF_KW type_exp
    |   ARRAY_KW LBRACKET open_array_index COMMA fixed_array_indices
            RBRACKET OF_KW type_exp
    ;
fixed_array_type_exp:
        ARRAY_KW LBRACKET fixed_array_indices RBRACKET OF_KW type_exp
    ;

open_array_index:
        const_exp DOTDOT STAR
    ;
fixed_array_indices:
        fixed_array_index
    |   fixed_array_indices COMMA fixed_array_index
    ;
```



```
fixed_array_index:
        simple_type_exp
    ;
ptr_type_exp:
        HAT simple_type_exp
    ;
set_type_exp:
        SET_KW OF_KW simple_type_exp
    ;
record_type_exp:
        RECORD_KW record_body END_KW
    ;
record_body:
        field_specs
    |   field_specs SEMI
    |   field_specs SEMI variant_dcl
    |   variant_dcl
    ;
id_colon_frob:
        IDENTIFIER COLON
    ;
variant_dcl:
        CASE_KW IDENTIFIER COLON simple_type_exp OF_KW union_components
        | id_colon_frob CASE_KW IDENTIFIER COLON simple_type_exp OF_KW
            union_components
    ;
field_specs:
        field_spec
    |   field_specs SEMI field_spec
    ;
field_spec:
        field_id_list attributed_type
    |   field_attrs field_id_list attributed_type
    ;
field_attrs:
        attribute_opener field_attr_list attribute_closer
    ;
field_attr_list:
        field_attr
    |   field_attr_list COMMA field_attr
    ;
```

```
field_attr:
        LAST_IS_KW LPAREN IDENTIFIER RPAREN
    |   MAX_IS_KW LPAREN IDENTIFIER RPAREN
    ;
field_id_list:
        id_colon_frob
    |   field_id COMMA field_id_list
    ;
field_id:
        IDENTIFIER
    ;
union_components:
        union_component
    |   union_components union_component
    ;

union_component:
        union_tag COLON LPAREN field_specs RPAREN SEMI
    |   union_tag COLON LPAREN field_specs SEMI RPAREN SEMI
    |   union_tag COLON LPAREN RPAREN SEMI
    |   union_tag COLON IDENTIFIER COLON LPAREN field_specs RPAREN SEMI
    |   union_tag COLON IDENTIFIER COLON LPAREN field_specs
            SEMI RPAREN SEMI
    |   union_tag COLON IDENTIFIER COLON LPAREN RPAREN SEMI
    ;
union_tag:
        tag
    |   union_tag COMMA tag
    ;
tag:
        const_exp
    ;
proc_ptr_type_exp:
        HAT PROCEDURE_KW parameter_list
    |   HAT PROCEDURE_KW
    ;
func_ptr_type_exp:
        HAT FUNCTION_KW parameter_list COLON type_exp
    ;
string0_type_exp:
        STRING0_KW LBRACKET const_exp RBRACKET
    ;
proc_def:
        proc_header IDENTIFIER SEMI proc_options
    |   proc_header IDENTIFIER parameter_list SEMI proc_options
    ;
proc_header:
        routine_attribute_list PROCEDURE_KW
    |   PROCEDURE_KW
    ;
```



```
proc_options:
        list_directed_options
    |   option_directed_options
    |   /* nothing */
    ;
list_directed_options:
        list_option_element
    |   list_directed_options list_option_element
    ;
list_option_element:
        IDENTIFIER SEMI
    ;
option_directed_options:
        OPTION_KW LPAREN options_list RPAREN SEMI
    ;
options_list:
        IDENTIFIER
    |   options_list COMMA IDENTIFIER
    ;
func_def:
        function_header IDENTIFIER COLON type_exp SEMI
    |   function_header IDENTIFIER parameter_list COLON type_exp SEMI
    ;
function_header:
        routine_attribute_list FUNCTION_KW ;
    |   FUNCTION_KW
    ;
routine_attribute_list:
        attribute_opener routine_attributes attribute_closer
    ;
routine_attributes:
        routine_attribute
    |   routine_attributes COMMA routine_attribute
    ;

routine_attribute:
    |   IDEMPOTENT_KW
    |   MAYBE_KW
    |   BROADCAST_KW
    ;
parameter_list:
        LPAREN parameters RPAREN
    ;
parameters:
        parameter_spec
    |   parameters SEMI parameter_spec
    |   /* nothing */
    ;
parameter_spec:
        parameter_ids COLON attributed_type
    ;
```

```
parameter_ids:
        parameter_attrs parameter_id_list
    ;
parameter_id_list:
        parameter_id
    |   parameter_id_list COMMA parameter_id
    ;
parameter_id:
        IDENTIFIER
    ;
parameter_attrs:
        parameter_class attribute_opener parameter_attr_list
                attribute_closer
    |   parameter_class
    ;
parameter_attr_list:
        parameter_attr
    |   field_attr
    |   parameter_attr_list COMMA parameter_attr
    |   parameter_attr_list COMMA field_attr
    ;
parameter_attr:
        COMM_STATUS_KW
    |   IN_KW
    |   OUT_KW
    |   REF_KW
    ;
parameter_class:
        IN_KW REF_KW
    |   IN_KW OUT_KW
    |   IN_KW
    |   OUT_KW
    ;

c_interface_tail:
        c_interface_marker c_interface_body c_interface_close
    ;
c_interface_marker:
        LBRACE
    ;
c_interface_close:
        RBRACE
    ;
c_attribute_opener:
        LBRACKET
    ;

c_attribute_closer:
        RBRACKET
    ;
```



```
c_interface_body:
        c_exports
    |   c_imports c_exports
    |   /* nothing */
    ;
c_imports:
        c_import
    |   c_imports c_import
    ;
c_import:
        IMPORT_KW STRING SEMI
    ;
c_exports:
        c_export
    |   c_exports  c_export
    ;

c_export:
        c_type_dcl      SEMI
    |   c_const_dcl     SEMI
    |   c_operation_dcl SEMI
    ;

c_const_dcl:
        CONST_KW c_type_spec IDENTIFIER EQUAL c_const_exp
    ;
c_const_exp:
        INTEGER_NUMERIC
    |   IDENTIFIER
    |   STRING
    |   NIL_KW
    |   TRUE_KW
    |   FALSE_KW
    ;
c_type_dcl:
        TYPEDEF_KW c_type_declarator
    ;
c_type_declarator:
        c_attributed_type_spec c_declarators
    |   c_type_spec c_declarators
    ;
c_attributed_type_spec:
        c_attribute_opener c_type_attributes c_attribute_closer
c_type_spec
    ;
c_type_attributes:
        c_type_attribute
    |   c_type_attributes COMMA c_type_attribute
    ;
```

```
c_type_attribute:
        LAST_IS_KW LPAREN IDENTIFIER RPAREN
    |   MAX_IS_KW LPAREN IDENTIFIER RPAREN
    |   HANDLE_KW
    |   TRANSMIT_AS_KW LPAREN IDENTIFIER RPAREN
    ;
c_type_spec:
        c_simple_type_spec
    |   c_constructed_type_spec
    ;
c_simple_type_spec:
        c_floating_point_type_spec
    |   c_integer_type_spec
    |   c_char_type_spec
    |   c_boolean_type_spec
    |   c_byte_type_spec
    |   c_void_type_spec
    |   c_named_type_spec
    |   c_handle_type_spec
    |   c_drep_type_spec
    ;
c_constructed_type_spec:
        c_struct_type_spec
    |   c_union_type_spec
    |   c_enum_type_spec
    |   c_set_type_spec
    |   c_string0_type_spec
    ;

c_named_type_spec:
        IDENTIFIER
    ;
c_floating_point_type_spec:
        FLOAT_KW
    |   DOUBLE_KW
    ;

c_integer_size_spec:
        SMALL_KW
    |   SHORT_KW
    |   LONG_KW
    |   HYPER_KW
    ;
c_integer_modifiers:
        c_integer_size_spec
    |   UNSIGNED_KW
    |   UNSIGNED_KW c_integer_size_spec
    |   c_integer_size_spec UNSIGNED_KW
    ;
```



```
c_integer_type_spec:
        INT_KW
    |   c_integer_modifiers
    |   c_integer_modifiers INT_KW
    ;

c_char_type_spec:
        CHAR_KW
    ;
c_boolean_type_spec:
        BOOLEAN_KW
    ;
c_byte_type_spec:
        BYTE_KW
    ;
c_void_type_spec:
        VOID_KW
    ;
c_handle_type_spec:
        HANDLE_T_KW
    ;
c_struct_type_spec:
        STRUCT_KW c_struct_body
    ;
c_struct_body:
        LBRACE c_member_list RBRACE
    ;

c_union_type_spec:
        c_union_header LBRACE c_union_body RBRACE
    ;
c_union_header:
        UNION_KW SWITCH_KW LPAREN c_simple_type_spec IDENTIFIER RPAREN
    |   UNION_KW SWITCH_KW LPAREN c_simple_type_spec IDENTIFIER
          RPAREN IDENTIFIER
    ;
c_union_body:
        c_union_case
    |   c_union_body c_union_case
    ;
c_union_case:
        c_union_case_list c_member
    ;
c_union_case_list:
        c_union_case_tag
    |   c_union_case_list c_union_case_tag
    ;
```

```
c_union_case_tag:
        CASE_KW c_const_exp COLON
    ;
c_member_list:
        c_member
    |   c_member_list c_member
    ;

c_member:
        c_attributed_type_spec c_member_attribute_list
           c_declarators SEMI
    |   c_attributed_type_spec  c_declarators SEMI
    |   c_type_spec c_member_attribute_list c_declarators SEMI
    |   c_type_spec c_declarators SEMI
    ;
c_member_attribute_list:
        c_attribute_opener c_member_attributes c_attribute_closer
    ;
c_member_attributes:
        c_member_attribute
    |   c_member_attributes COMMA c_member_attribute
    ;
c_member_attribute:
        LAST_IS_KW LPAREN IDENTIFIER RPAREN
    |   MAX_IS_KW LPAREN IDENTIFIER RPAREN
    ;

c_enum_type_spec:
        ENUM_KW c_enum_body
    |   LONG_KW ENUM_KW c_enum_body
    |   SHORT_KW ENUM_KW c_enum_body
    ;
c_enum_body:
        LBRACE c_enum_ids RBRACE
    ;
c_enum_ids:
    |   c_enum_id
    |   c_enum_ids COMMA c_enum_id
    ;
c_enum_id:
        IDENTIFIER
    ;
c_set_type_spec:
        BITSET_KW c_type_spec
    |   LONG_KW BITSET_KW c_type_spec
    |   SHORT_KW BITSET_KW c_type_spec
    ;
c_string0_type_spec:
        STRING0_KW LBRACKET c_const_exp RBRACKET
    ;
```



```
c_declarators:
        c_declarator
    |   c_declarators COMMA c_declarator
    ;

c_declarator:
        c_simple_declarator
    |   c_complex_declarator
    ;

c_simple_declarator:
        IDENTIFIER
    ;
c_complex_declarator:
        c_pointer_declarator
    |   c_array_declarator
    |   c_function_ptr_declarator
    |   c_reference_declarator
    ;
c_pointer_declarator:
        STAR IDENTIFIER
    |   STAR CONST_KW IDENTIFIER
    |   CONST_KW STAR IDENTIFIER
    ;
c_reference_declarator:
        AMPER IDENTIFIER
    |   AMPER CONST_KW IDENTIFIER
    |   CONST_KW AMPER IDENTIFIER
    ;

c_array_declarator:
        IDENTIFIER LBRACKET RBRACKET
    |   IDENTIFIER LBRACKET STAR RBRACKET
    |   IDENTIFIER LBRACKET RBRACKET c_fixed_array_indices
    |   IDENTIFIER LBRACKET STAR RBRACKET c_fixed_array_indices
    |   IDENTIFIER c_fixed_array_indices
    ;
c_fixed_array_indices:
        c_fixed_array_index
    |   c_fixed_array_indices c_fixed_array_index
    ;
c_fixed_array_index:
        LBRACKET const_exp RBRACKET
    ;
c_function_ptr_declarator:
        c_function_ptr_hdr c_parameter_[illegible]
    ;
c_function_ptr_hdr:
        LPAREN STAR IDENTIFIER RPAREN
    ;
```

```
c_operation_dcl:
        c_routine_attribute_list c_simple_type_spec IDENTIFIER
           c_parameter_dcls
    |   c_simple_type_spec IDENTIFIER  c_parameter_dcls
    |   IDENTIFIER  c_parameter_dcls
    ;
c_routine_attribute_list:
        c_attribute_opener c_routine_attributes c_attribute_closer
    ;
c_routine_attributes:
        c_routine_attribute
    |   c_routine_attributes COMMA c_routine_attribute
    ;
c_routine_attribute:
    |   IDEMPOTENT_KW
    |   MAYBE_KW
    |   BROADCAST_KW
    ;

c_parameter_dcls:
        LPAREN c_param_list RPAREN
    ;
c_param_list:
        c_param_dcl
    |   c_param_list COMMA c_param_dcl
    |   /* nothing */
    ;
c_param_dcl:
        c_attributed_type_spec c_param_attribute_list c_declarator
    |   c_attributed_type_spec  c_declarator
    |   c_type_spec c_param_attribute_list c_declarator
    ;
c_param_attribute_list:
        c_attribute_opener c_param_attributes c_attribute_closer
    ;
c_param_attributes:
        c_param_attribute
    |   c_member_attribute
    |   c_param_attributes COMMA c_param_attribute
    |   c_param_attributes COMMA c_member_attribute
    ;
c_param_attribute:
        IN_KW
    |   OUT_KW
    |   COMM_STATUS_KW
    ;
%%
```

Annexe D:

Network Data Representation (NDR)

Chapter 9

Network Data Representation

Most application programs treat inputs and outputs as structured values such as integers, arrays, and pointers; one role of NIDL is to provide syntax for describing these structured values. However, the NCA/RPC protocol specifies that inputs and outputs be passed in byte streams; the role of NDR is to support the mapping of NIDL data types onto byte streams. NDR defines scalar data types, constructed data types, and representations for these types in a byte stream.

For some scalar data types, NDR defines several data representation formats. For example, NDR defines ASCII and EBCDIC formats for characters. When a client or server sends an NCA/RPC packet, the formats used are determined dynamically and are identified in the format label of the packet. The data representation formats and the format label support the NCA multicanonical approach to data conversion.

This chapter describes [insert attenuated rep]

- The set of NDR scalar types and the supported data representation formats for these types
- The set of NDR constructed types
- The ordering of parameter representations in NDR input and output streams
- The NDR format label

9.1 NDR Scalar Types

NDR defines a set of scalar data types to represent boolean values, characters, four sizes of signed and unsigned integers, two sizes of floating-point numbers, and uninterpreted bytes.

For characters, integers, and floating-point numbers, NDR defines more than one representation format. The formats used in an NCA/RPC packet are identified in the NDR format label.

All NDR scalar data types are multiples of bytes in length. A byte is eight bits.

9.1.1 Conventions

The figures in this section adopt the following conventions:

- In depicting data types larger than one byte, we order the bytes from top to bottom. The topmost byte appears first in the byte stream and the bottommost byte appears last.
- We do not show alignment gaps, which can appear in the byte stream before an item. See [].
- In depicting data types with bit fields, we place the most significant bit at the left and the least significant bit at the right.
- We use the abbreviations "MSB" for "most significant bit" and "LSB" for "least significant bit".

9.1.2 Booleans

A boolean is a logical quantity that assumes one of two values: "true" or "false". NDR represents a boolean as one byte. It represents a value of "false" as a zero byte and a value of "true" as a nonzero byte.

Figure 9–1 illustrates the boolean data type as it appears in the byte stream.

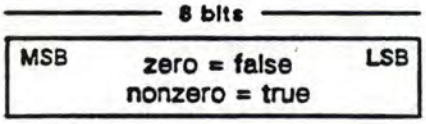

Figure 9–1. Boolean Data Type

9.1.3 Characters

NDR represents a character as one byte. Characters have two representation formats: ASCII and EBCDIC. Appendix D provides an ASCII/EBCDIC conversion chart.

Figure 9-2 illustrates the character type as it appears in the byte stream.

Figure 9-2. Character Data Type

9.1.4 Integers

NDR defines both signed and unsigned integers in four sizes:

- small — an 8-bit integer, represented in the byte stream as one byte
- short — a 16-bit integer, represented in the byte stream as two bytes
- long — a 32-bit integer, represented in the byte stream as four bytes
- hyper — a 64-bit integer, represented in the byte stream as eight bytes

NDR represents signed integers in two's complement notation and represents unsigned integers as unsigned binary numbers. There are two integer formats: big-endian and little-endian. If the integer format is big-endian, the bytes of the representation are ordered in the byte stream from the most significant byte to the least significant byte. If the integer format is little-endian, the bytes of the representation are ordered in the byte stream from the least significant to the most significant.

Figure 9–3 illustrates the integer types in big-endian and little-endian format.

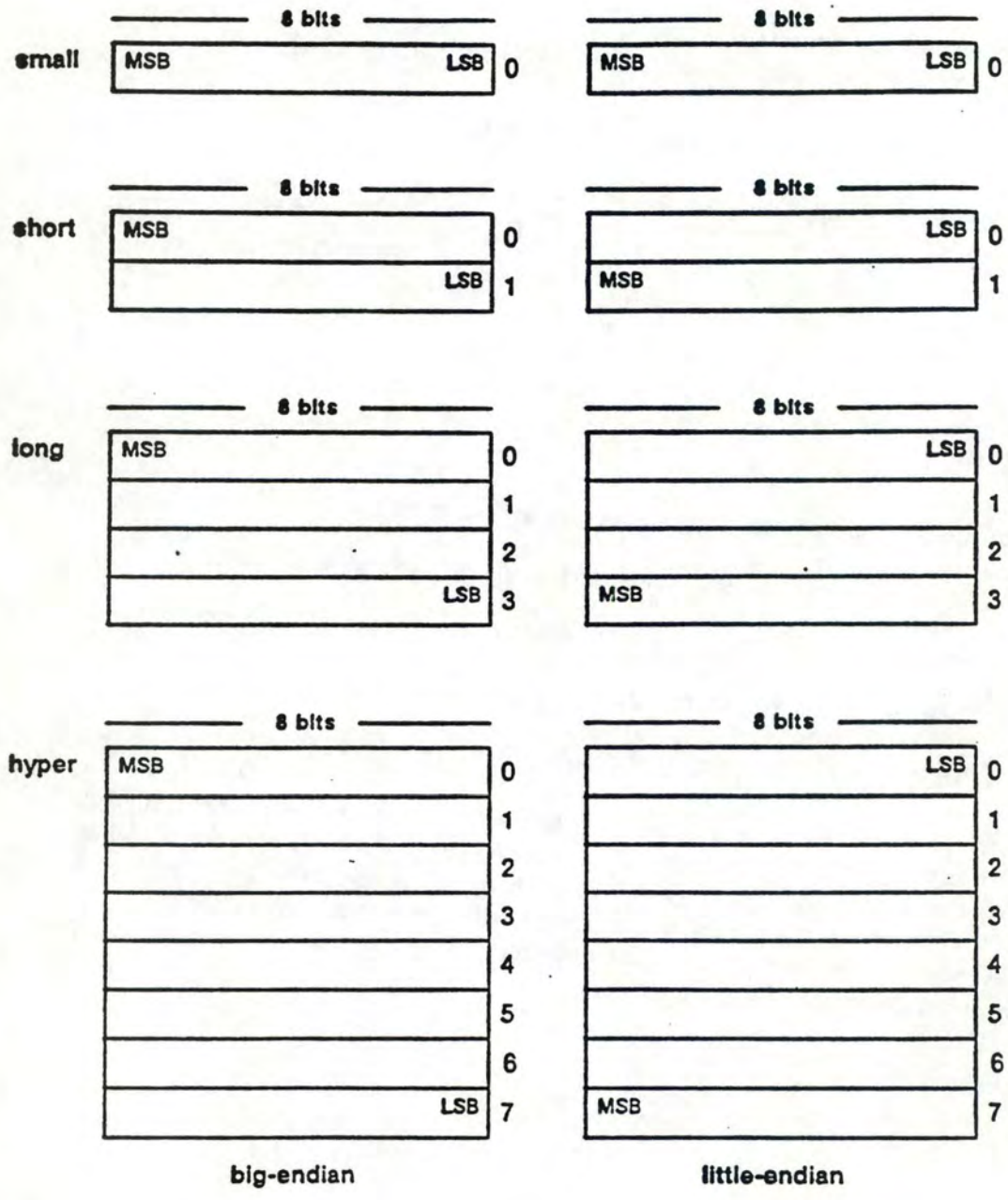

Figure 9–3. Big-Endian and Little-Endian Integer Formats

9.1.5 Floating-Point Numbers

NDR defines single-precision and double-precision floating-point data types. It represents single precision in four bytes and double precision in eight bytes.

NDR supports the following floating-point data representation formats for single-precision and double-precision floating point:

- IEEE single and double floating point, which comply with IEEE standard 754.
- VAX F_floating and G_floating formats as defined in the *VAX11 Architecture Handbook*, Copyright 1979, Digital Equipment Corporation. [Or *VAX Technical Summary*, 1980?]
- Cray floating-point format, as defined in the documentation produced by Cray Research, Inc. [Specific citation?]
- IBM short and long formats, as defined in the *IBM System/370 Principles of Operation*, Copyright 1974, International Business Machines Corporation.

Table 9–1 is a conversion chart that shows how NDR single-precision and double-precision floating-point types correspond to the supported floating-point formats.

Table 9–1. NDR Floating-Point Conversions

NDR Values	Conversion Values			
	IEEE	VAX	Cray	IBM
single	single (4 bytes)	F (4 bytes)	single (4 bytes)	short (4 bytes)
double	double (8 bytes)	G (8 bytes)	double (8 bytes)	long (8 bytes)

The representation of a floating-point number comprises three fields:

- The sign bit, which indicates the sign of the number. Values 0 and 1 represent positive and negative, respectively. This field is one bit in length.
- The exponent of the number (base 16 in IBM format, base 2 in all others), biased by an excess. The size of this field varies according to the format, as does the excess.
- The fractional part of the number's mantissa (base 16 in IBM format, base 2 in all others). This field is also called the number's coefficient. The size of this field varies according to the format [as does the normalization of the mantissa].

The remainder of this subsection describes how floating-point numbers are represented in the byte stream in IEEE, VAX, Cray, and IBM formats.

9.1.5.1 IEEE Format

Single IEEE floating-point format is 32 bits in length, consisting of a 1-bit sign, an 8-bit exponent field (excess 127) and a 23-bit mantissa that represents a fraction in the range 1.0 (inclusive) to 2.0 (exclusive). Double IEEE floating-point format is 64 bits in length with a 1-bit sign, an 11-bit exponent (excess 1023), and a 52-bit mantissa.

IEEE floating-point numbers are used on machines made by a variety of manufacturers and based on a variety of architectures. Some of these machines (for example, those based on Intel 80x86 processors) use little-endian representation for integers; other machines (for example, those based on Motorola MC680x0 processors) use big-endian representation. When a recipient interprets an NDR byte stream whose format label specifies IEEE floating-point format, it uses the integer representation in the format label to determine the byte order of the IEEE floating-point number.

Figure 9–4 and Figure 9–5 illustrate IEEE single-precision and double-precision floating-point format in big-endian and little-endian integer representation.

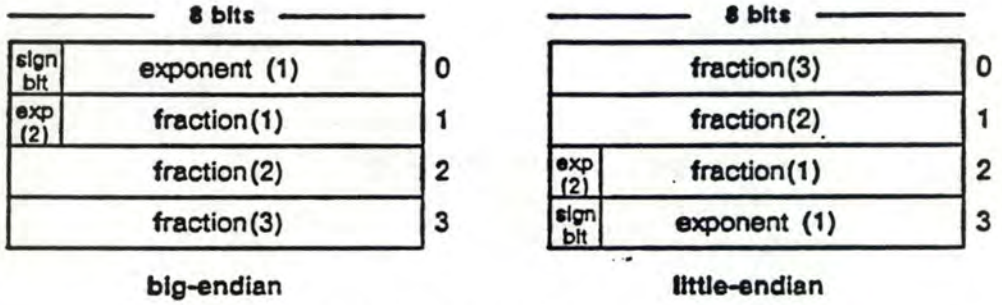

Figure 9–4. IEEE Single-Precision Floating-Point Format

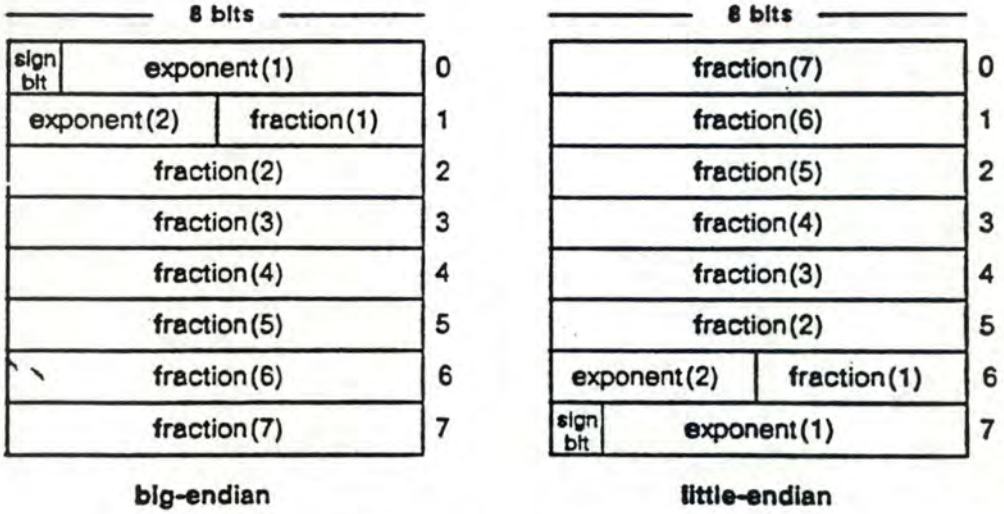

Figure 9–5. IEEE Double-Precision Floating-Point Format

9.1.5.2 VAX Format

VAX architecture defines four floating-point formats: F_floating, D_floating, G_floating, and H_floating. F_floating format is 32 bits in length, including a 1-bit sign, an 8-bit exponent (excess 128), and a 23-bit mantissa that represents a fraction in the range 0.5 (inclusive) to 1.0 (exclusive). D_floating format is 64 bits, with a 1-bit sign, an 8-bit exponent, and a 56-bit mantissa. G_floating format is also 64 bits, with a 1-bit sign, an 11-bit exponent (excess 1024), and a 52-bit mantissa. H_floating format is 128 bits.

Although the VAX architecture supports four floating-point formats, NDR uses only VAX F_floating format to represent VAX single-precision floating-point numbers and VAX G_floating format to represent VAX double-precision floating-point numbers.

Figure 9–6 and Figure 9–7 illustrate VAX F and G floating-point representations as they appear in the byte stream.

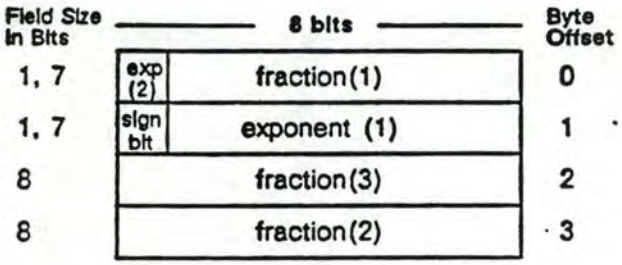

Figure 9–6. VAX Single-Precision (F) Floating-Point Format

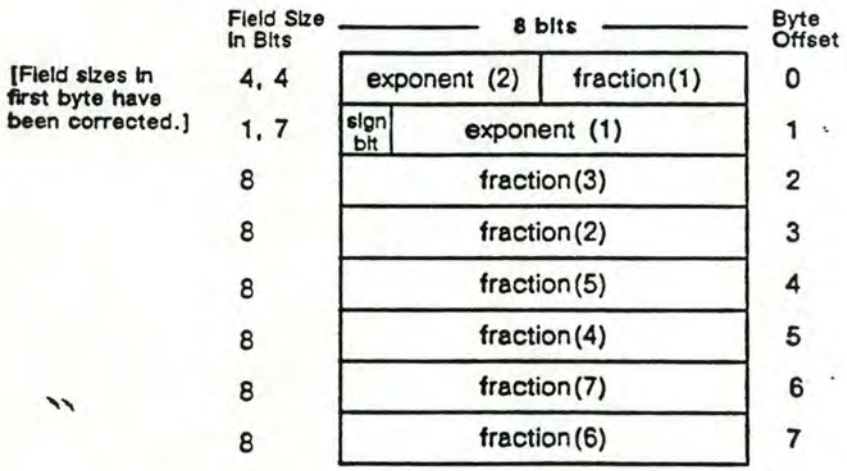

Figure 9–7. VAX Double-Precision (G) Floating-Point Format

Table 9–2 defines the fields in Figure 9–6 in VAX architecture terms.

Table 9–2. VAX F Floating-Point Fields

Field Name	Bit Field
Exponent(1)	8:14
Exponent(2)	7:7
Sign Bit	15:15
Fraction(1)	0:6
Fraction(2)	24:31
Fraction(3)	16:23

Figure 9–6 and Figure 9–7 illustrate VAX F and G floating-point format as they appear on VAX machines, which use a little-endian representation for integers. However, some machines may implement VAX floating-point format with a big-endian representation. When a recipient interprets an NDR byte stream whose format label specifies VAX floating-point format, it uses the integer representation in the format label to determine the byte order of the floating-point number.

9.1.5.3 Cray Format

Cray machine architecture defines only a double-precision floating-point format. However, because Cray machines may be required to handle single-precision floating-point values (for instance, if single-precision values are specified in an interface definition), NDR defines a single-precision floating-point format for Cray machines; this format is identical to IEEE big-endian short format.

A Cray double-precision floating-point number is 64 bits in length and consists of a 1-bit sign, a 15-bit exponent (16384 excess), and a 48-bit fraction. [Range for fraction?] A Cray single-precision floating-point number is 32 bits in length and consists of a 1-bit sign, an 8-bit exponent (excess 127), and a 23-bit mantissa that represents a fraction in the range 1.0 (inclusive) to 2.0 (exclusive).

Figure 9–8 illustrates the Cray floating-point formats.

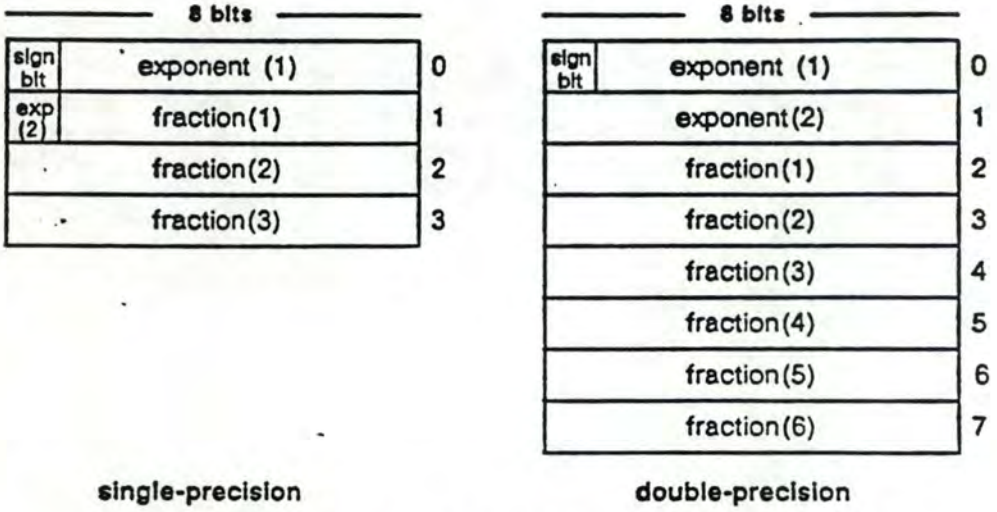

Figure 9–8. Cray Floating-Point Formats

9.1.5.4 IBM Format

The IBM [System/360 and System/370] architecture defines short and long floating-point formats for single-precision and double-precision floating-point values, respectively. An IBM short floating-point number consists of a 1-bit sign, a 7-bit exponent, and a 24-bit fraction. An IBM long floating-point number consists of a 1-bit sign, a 7-bit exponent, and a 56-bit fraction. [Ranges for fractions?] The IBM formats represent both the exponent and the fraction in hexadecimal rather than binary notation. Consequently, the exponent is base 16, while the fraction is composed of either six 4-bit hexadecimal digits or fourteen 4-bit hexadecimal digits.

Figure 9–9 illustrates the IBM short and long floating-point formats.

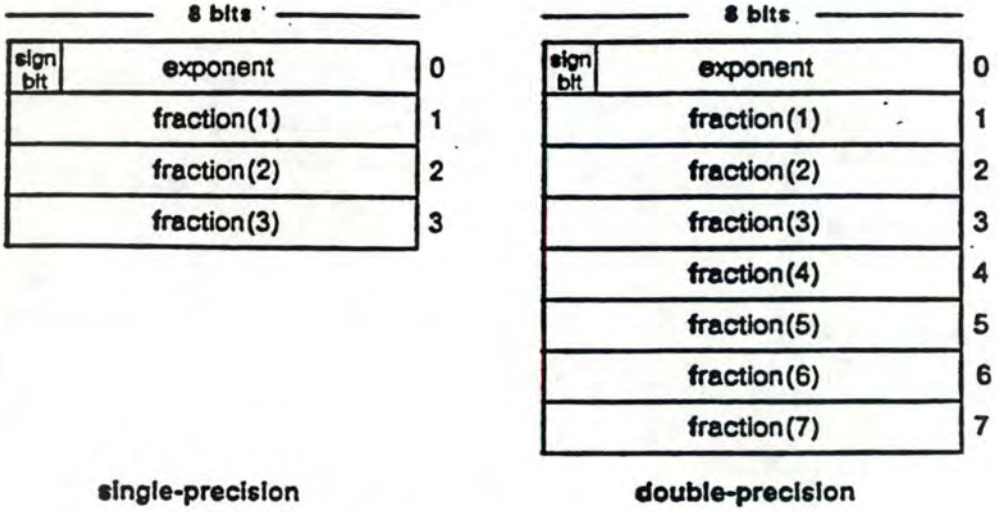

Figure 9–9. IBM Floating-Point Format

Figure 9–9 illustrates IBM floating-point format as it appears on IBM 360 machines, which use a big-endian representation for integers [and floating-point numbers?]. However, some machines may implement IBM floating-point with a little-endian representation. When a recipient interprets an NDR byte stream whose format label specifies IBM floating-point format, it uses the integer representation in the format label to determine the byte order of the floating-point number.

Note that IBM PCs use IEEE single-precision format rather than IBM short format.

9.1.6 Uninterpreted Bytes

NDR defines an uninterpreted byte data type for which no internal format is defined and on which no format conversions are made.

9.2 NDR Constructed Types

NDR supports data types that are constructed from the NDR scalar data types described in the previous section. The NDR constructed types include arrays, strings, structures, unions, variant structures, pipes, and pointers.

NDR represents every NDR constructed type as a sequence of NDR scalar values. The representation formats for these scalar values are identified in the NDR format label.

All NDR constructed data types are multiples of bytes in length.

9.2.1 Conventions

The figures in this section adopt the following conventions:

- We order data from left to right. The leftmost item appears first in the byte stream and the rightmost item appears last.
- We do not show alignment gaps, which can appear in the byte stream either before or within an item. See [].
- We use ellipsis points between items labeled "first" and "last" to indicate that any number of items can appear in the byte stream. Unless we state otherwise, at least one item, which would be both first and last, must appear. We use ellipsis points between items labeled "first" and "penultimate" similarly. Figure 9–10 gives one example of this notation.
- We use braces and arrows to indicate an item whose composition is exploded in another part of the figure. Figure 9–15 gives one example of this notation.
- We use the abbreviations "max" for "maximum" and "rep" for "representation."

9.2.2 Arrays

An array is an ordered, indexed collection of homogeneous elements. The elements of an array can be of any NDR scalar or constructed type except [pipes? small arrays? conformant/varying arrays/structures??].

NDR defines several representations for arrays. The representation used depends on

- Whether the array is unidimensional or multidimensional
- Whether the array is conformant
- Whether the array is varying

NDR defines special representations for arrays that contain pointers (described in []) and for structures that contain a conformant array (described in []).

For backward compatibility with previous versions of NCA, NDR supports several small array types, which we describe in []. Where necessary for clarity, we refer to the arrays described here in this subsection as large arrays.

9.2.2.1 Unidimensional Fixed Arrays

A fixed array is an array that is neither conformant nor varying. In a fixed array, the maximum number of elements is known a priori, and every element is passed in every call.

NDR represents a fixed array as an ordered sequence of representations of the array elements.

Figure 9–10 illustrates a fixed array as it appears in the byte stream.

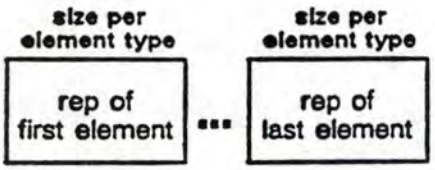

Figure 9–10. Unidimensional Fixed Array Representation

9.2.2.2 Unidimensional Conformant Arrays

A conformant array is an array in which the maximum number of elements is not known a priori and therefore is included in the representation of the array.

NDR represents a conformant array as an ordered sequence of representations of the array elements, preceded by an unsigned long integer. The integer gives the maximum number of elements in the array.

A conformant array can contain at most $2^{32} - 1$ elements.

Figure 9–11 illustrates a conformant array as it appears in the byte stream.

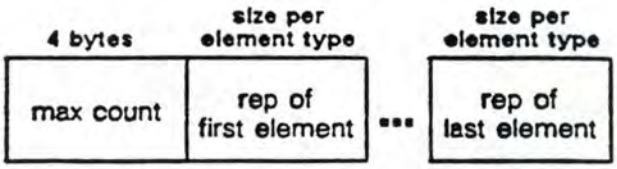

Figure 9–11. Unidimensional Conformant Array Representation

9.2.2.3 Unidimensional Varying Arrays

A varying array is an array in which the actual number of elements passed in a given call varies and therefore is included in the representation of the array.

NDR represents a varying array as an ordered sequence of representations of the array elements, preceded by two unsigned long integers. The first integer gives the offset from the first index of the array to the first index of the actual subset being passed. The second integer gives the actual number of elements being passed.

A varying array can contain at most $2^{32}-1$ elements.

Figure 9–12 illustrates a varying array as it appears in the byte stream.

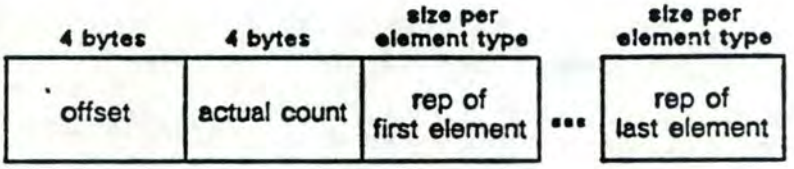

Figure 9–12. Unidimensional Varying Array Representation

9.2.2.4 Unidimensional Conformant and Varying Arrays

An array can be both conformant and varying.

NDR represents a conformant and varying array as an ordered sequence of representations of the array elements, preceded by three unsigned long integers. The first integer gives the maximum number of elements in the array. The second integer gives the offset from the first index of the array to the first index of the actual subset being passed. The third integer gives the actual number of elements being passed.

A conformant and varying array can contain at most $2^{32}-1-o$ elements, where o is the offset.

Figure 9–13 illustrates a conformant and varying array as it appears in the byte stream.

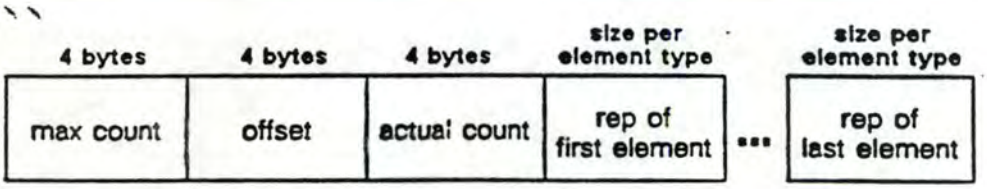

Figure 9–13. Unidimensional Conformant and Varying Array Representation

9.2.2.5 Ordering of Elements in Multidimensional Arrays

NDR orders multidimensional array elements so that the index of the first dimension varies slowest and the index of the last dimension varies fastest.

For example, consider an array in two dimensions with indexes ranging from 0 to 1 in the first dimension and from 0 to 2 in the second dimension. NDR orders the elements of the array as follows:

A(0, 0), A(0, 1), A(0, 2), A(1, 0), A(1, 1), A(1, 2)

where the notation A(*i*, *j*) denotes the element with index *i* in the first dimension and index *j* in the second dimension.

9.2.2.6 Multidimensional Fixed Arrays

A multidimensional array is fixed if in all of its dimensions, the maximum number of elements is known a priori, and every element is passed in every call.

NDR represents fixed multidimensional arrays in the same format as fixed unidimensional arrays, as shown in Figure 9–14.

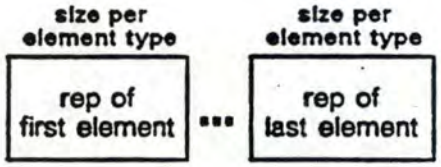

Figure 9–14. Multidimensional Fixed Array Representation

9.2.2.7 Multidimensional Conformant Arrays

A multidimensional array is conformant if the maximum size in any of its dimensions is not known a priori.

NDR represents a multidimensional conformant array as an ordered sequence of unsigned long integers, followed by an ordered sequence of representations of the array elements. The integers give the maximum size in each dimension of the array.

A multidimensional conformant array can span at most $2^{32} - 1$ elements in each dimension.

Figure 9–15 illustrates a multidimensional conformant array as it appears in the byte stream.

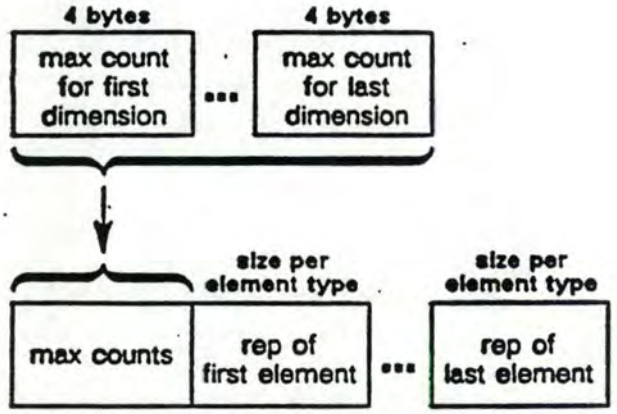

Figure 9–15. Multidimensional Conformant Array Representation

9.2.2.8 Multidimensional Varying Arrays

A multidimensional array is varying if the actual size in any of its dimensions varies.

NDR represents a multidimensional varying array as an ordered sequence of pairs of unsigned long integers, followed by an ordered sequence of representations of the array elements. There is one integer pair for each dimension of the array. The first integer gives the offset from the first index in the dimension to the first index of the subset being passed. The second integer gives the actual size in the dimension for the subset being passed.

A multidimensional varying array can span at most $2^{32} - 1$ elements in each dimension.

Figure 9–16 illustrates a multidimensional varying array as it appears in the byte stream. The offsets and actual counts iterate pairwise.

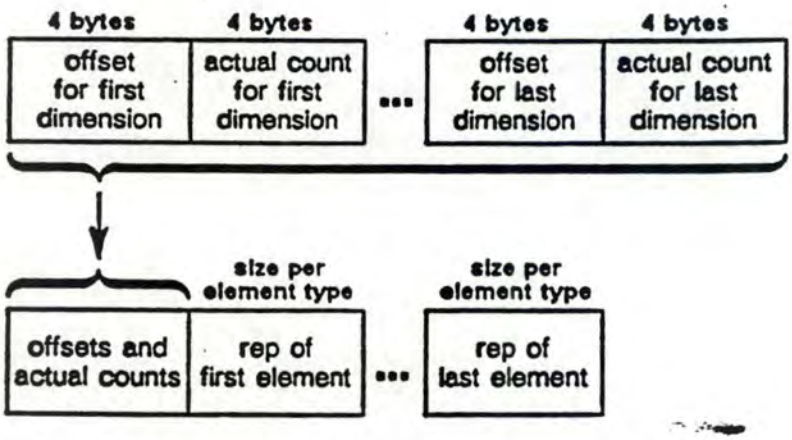

Figure 9–16. Multidimensional Varying Array Representation

9.2.2.9 Multidimensional Conformant and Varying Arrays

A multidimensional array can be both conformant and varying.

NDR represents a multidimensional conformant and varying array as an ordered sequence of unsigned long integers, followed by an ordered sequence of pairs of unsigned long integers, followed by an ordered sequence of representations of the array elements. In the sequence of integers, there is one integer for each dimension of the array, and the integers give the maximum size in each dimension. In the sequence of pairs of integers, there is one pair of integers for each dimension of the array; the first integer gives the offset from the first index in the dimension to the first index of the subset being passed, and the second integer gives the actual size in the dimension for the subset being passed.

Each dimension of a multidimensional conformant and varying array can span at most $2^{32} - 1 - o$ elements, where o is the offset in that dimension.

Figure 9–17 illustrates a multidimensional conformant and varying array as it appears in the byte stream. The offsets and actual counts iterate pairwise.

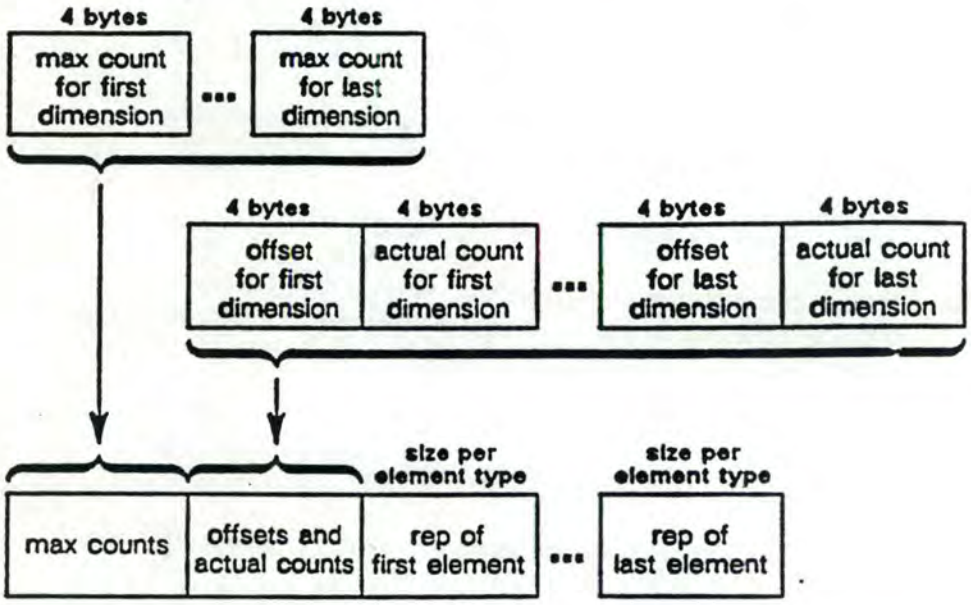

Figure 9–17. Multidimensional Conformant and Varying Array Representation

9.2.3 Small Arrays

For backward compatibility with previous versions of NCA, NDR defines small varying arrays and small conformant and varying arrays. The elements of a small array can be of any NDR scalar or constructed type except pipes, conformant and/or varying arrays (large or small), and structures that contain conformant and/or varying arrays (large or small).

NDR defines special representations for structures that contain a small varying array or a small conformant and varying array (described in []).

9.2.3.1 Small Unidimensional Varying Arrays

NDR represents a small varying array as an ordered sequence of representations of the array elements, preceded by an unsigned short integer giving the actual number of elements being passed.

A small varying array can contain at most $2^{16} - 1$ elements.

Figure 9–18 illustrates a small varying array as it appears in the byte stream.

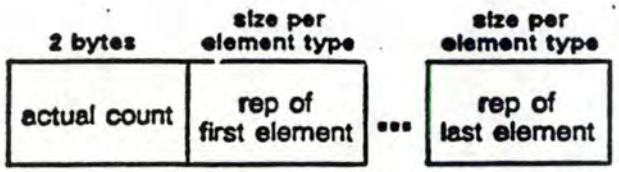

Figure 9–18. Small Unidimensional Varying Array Representation

9.2.3.2 Small Unidimensional Conformant and Varying Arrays

NDR represents a small conformant and varying array as an ordered sequence of representations of the array elements, preceded by two unsigned short integers. The first integer gives the maximum number of elements in the array. The second integer gives the actual number of elements being passed.

A small conformant and varying array can contain at most $2^{16}-1$ elements.

Figure 9–19 illustrates a small conformant and varying array as it appears in the byte stream.

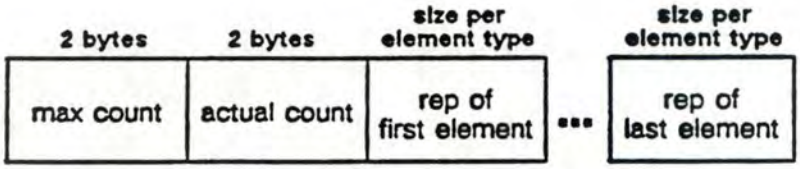

Figure 9–19. Small Unidimensional Conformant and Varying Array Representation

9.2.3.3 Ordering of Elements in Small Multidimensional Arrays

As with large multidimensional arrays, the elements of a small multidimensional array are ordered so that the index of the first dimension varies slowest and the index of the last dimension varies fastest.

See the example in [].

9.2.3.4 Small Multidimensional Varying Arrays

NDR represents small multidimensional varying arrays in the same format as small unidimensional varying arrays, as shown in Figure 9–20.

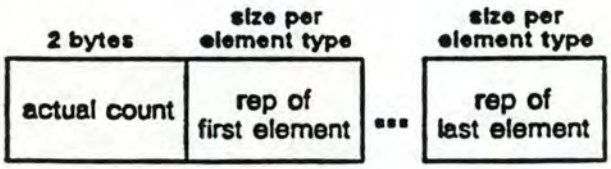

Figure 9–20. Small Multidimensional Varying Array Representation

Only the first dimension of a small array can be varying; other dimensions must be fixed. The actual count is for the entire array, not for the first dimension.

A small multidimensional varying array can contain at most $2^{16}-1$ elements.

9.2.3.5 Small Multidimensional Conformant and Varying Arrays

NDR represents small multidimensional conformant and varying arrays in the same format as small unidimensional conformant and varying arrays, as shown in Figure 9–21.

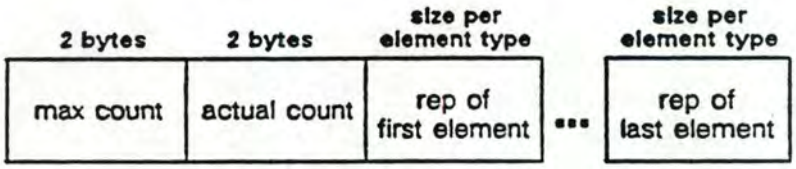

Figure 9–21. Small Multidimensional Conformant and Varying Array Representation

Only the first dimension of a small array can be conformant and varying; other dimensions must be fixed. The maximum and actual counts are for the entire array, not for the first dimension.

A small multidimensional conformant and varying array can contain at most $2^{16}-1$ elements.

9.2.4 Zero-Terminated Strings

A zero-terminated string is an ordered collection of characters terminated by a null character.

NDR represents a zero-terminated string as an unsigned short integer giving the number of non-null characters in the string, followed by representations of the non-null characters in the string, followed by a zero byte.

A zero-terminated string can contain at most $2^{16} - 1$ non-null characters.

Figure 9–22 illustrates a zero-terminated string as it appears in the byte stream.

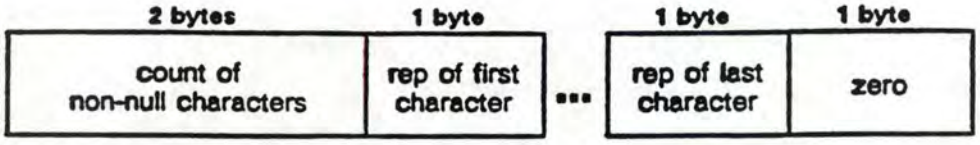

Figure 9–22. Zero-Terminated String Representation

9.2.5 Structures

A structure is an ordered collection of possibly heterogeneous members. A structure member can be of any NDR scalar or constructed type. However, a conformant array (large or small) can appear in a structure only as the last member, and a structure that contains a conformant array (large or small) can appear in another structure only as the last member.

NDR represents a structure as an ordered sequence of representations of the structure members.

Figure 9–23 illustrates a structure as it appears in the byte stream.

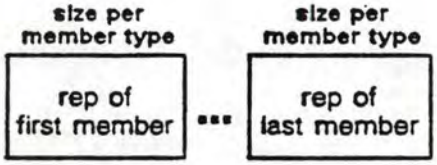

Figure 9–23. Structure Representation

NDR defines special representations for structures that contain a conformant array (described in []) and for structures that contain pointers (described in []).

9.2.6 Structures Containing Arrays

NDR defines special representations for a structure that contains a conformant array or a conformant and varying array.

9.2.6.1 Structures Containing a Conformant Array

A structure can contain a conformant array only as its last member.

In the NDR representation of a structure that contains a conformant array, the unsigned long integers that give maximum element counts for dimensions of the array are promoted to the beginning of the structure, and the array elements appear in place at the end of the structure. If a structure that contains a conformant array is itself a member of another structure, the maximum element counts are further promoted to the beginning of the embedding structure. Promotion of the maximum counts iterates through any other consecutively embedding structures.

The maximum element counts are promoted so that, when necessary, a recipient of the structure can use the counts to determine how much storage to allocate for the structure.

Figure 9–24 illustrates a structure containing a conformant array as it appears in the byte stream.

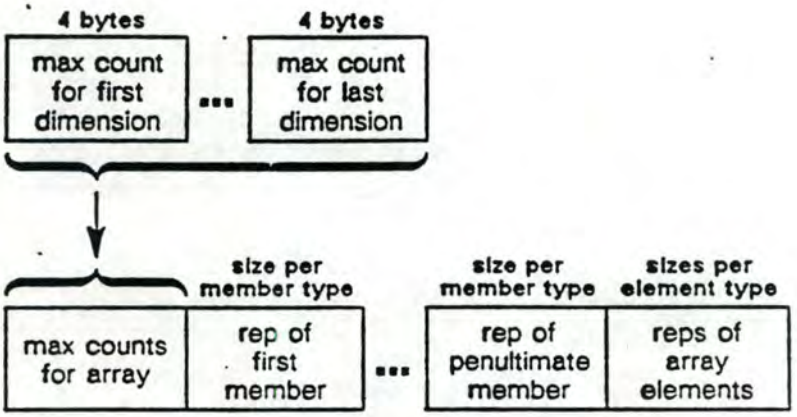

Figure 9–24. Representation of a Structure Containing a Conformant Array

A structure that contains a unidimensional conformant array has a degenerate form of this representation, with only one maximum count.

9.2.6.2 Structures Containing a Conformant and Varying Array

A structure can contain a conformant and varying array only as its last member.

In the NDR representation of a structure that contains a conformant and varying array, the maximum counts for dimensions of the array are promoted to the beginning of the structure, but the offsets and actual counts remain in place at the end of the structure, immediately preceding the array elements. If a structure that contains a conformant and varying array is itself a member of another structure, the maximum counts are further promoted to the beginning of the embedding structure. Promotion of the maximum counts iterates through any other consecutively embedding structures.

The maximum counts are promoted so that a recipient of the structure can use the counts to determine how much storage to allocate for the structure. The offsets and actual counts do not impinge on storage allocation and therefore are not promoted.

Figure 9–25 illustrates a structure containing a conformant and varying array as it appears in the byte stream. The offsets and actual counts iterate pairwise.

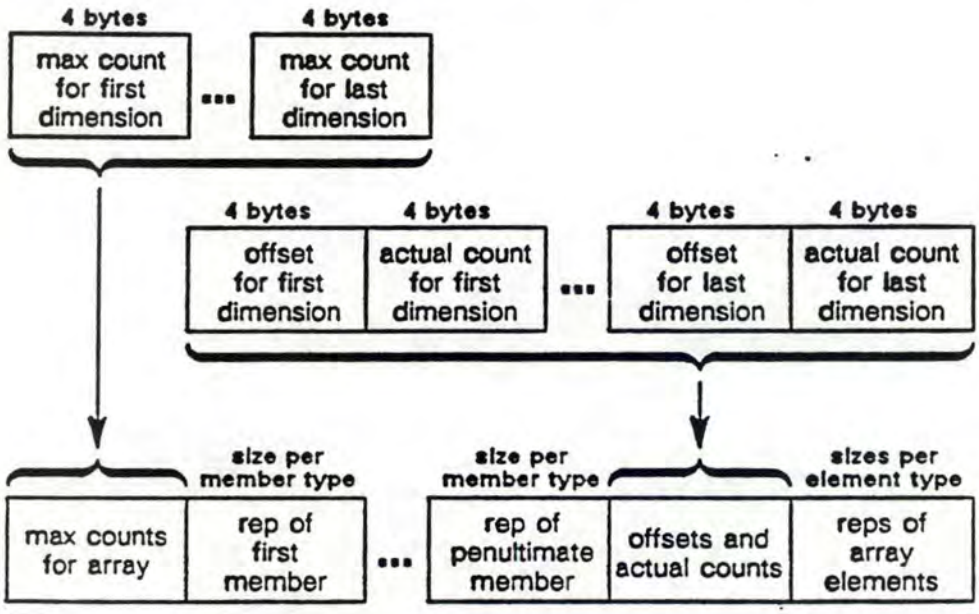

Figure 9–25. Representation of a Structure Containing a Conformant and Varying Array

A structure that contains a unidimensional conformant and varying array has one maximum count, one offset, and one actual count.

9.2.7 Structures Containing Small Arrays

For backward compatibility with previous versions of NCA, NDR defines representations for a structure that contains a small varying array or a small conformant and varying array.

Figure 9-23 illustrates a pipe as it appears in the byte stream.

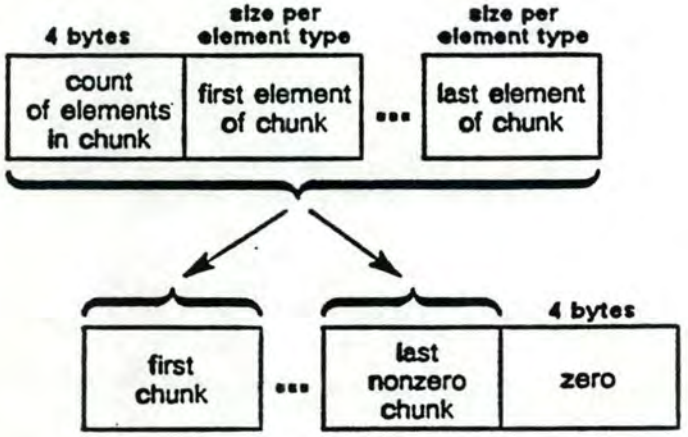

Figure 9-29. Pipe Representation

A pipe cannot be an element of another pipe, an element of an array, a member of a structure or variant structure, or a member of a union.

9.2.10 Pointers

NDR defines three classes of pointers that differ both in semantics and in representation:

- A "reference pointer" cannot be null and cannot be an alias.
- A "unique pointer" can be null but cannot be an alias.
- A "full pointer" can be null and can be an alias.

If a pointer points to nothing, it is null.

If the input and output byte streams pertaining to one remote procedure call contain several pointers that point to the same thing, the first of these pointers to be transmitted is considered primary and the others are considered aliases.

The scope of aliasing for a pointer extends to all streams transmitted in the service of one remote procedure call: any inputs in the request that initiates the call, any outputs in the response to the call, and any inputs and outputs associated with callbacks that occur during execution of the call. The headers of the packets in which these streams are transmitted all have the same sequence number.

Aliasing does not apply to null pointers.

We refer to pointers that are parameters in remote procedure calls as "top level pointers" and we refer to pointers that are elements of arrays, members of structures, or members of unions as "embedded pointers". NDR defines different representations for top level and embedded pointers. [] describes the NDR representation for top level pointers. [] describes the NDR representation for embedded pointers.

9.2.11 Top Level Pointers

This subsection describes the NDR representation for pointers that are parameters in remote procedure calls.

9.2.11.1 Top Level Full Pointers

NDR represents a null full pointer as an unsigned long integer with the value zero.

NDR represents the first instance in a byte stream of a non-null full pointer in two parts: the first part is a nonzero unsigned long integer that identifies the referent; the second part is the representation of the referent. NDR represents subsequent instances in the same byte stream of the same pointer only by the referent identifier.

Each referent in the input and output streams pertaining to one remote procedure call is associated with a referent identifier. A primary pointer and its aliases all have the same referent identifier. [Referent identifiers required to be unique? dense? sequential? Requirements extend to embedded pointers.]

Figure 9–30 illustrates the three possible representations for top level full pointers.

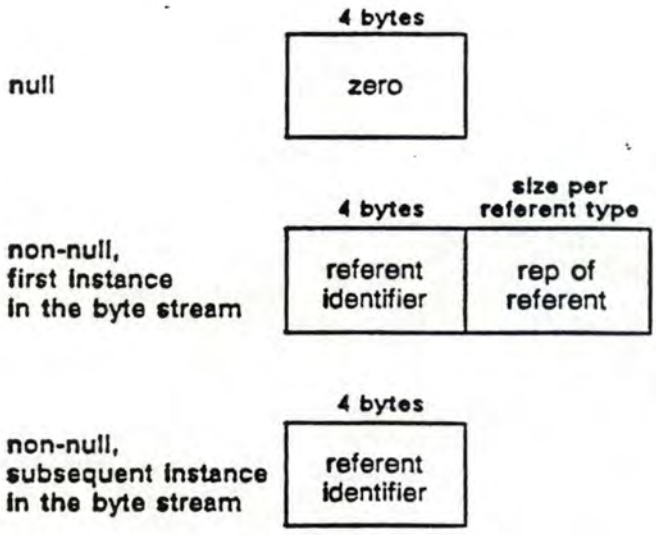

Figure 9–30. Top Level Full Pointer Representation

9.2.11.2 Top Level Unique Pointers

NDR represents a null unique pointer as an unsigned long integer with the value zero.

NDR represents a non-null unique pointer in two parts. The first part is an unsigned long integer with an arbitrary nonzero value. The second part is the representation of the referent.

Figure 9–31 illustrates top level null and non-null unique pointers as they appear in the byte stream.

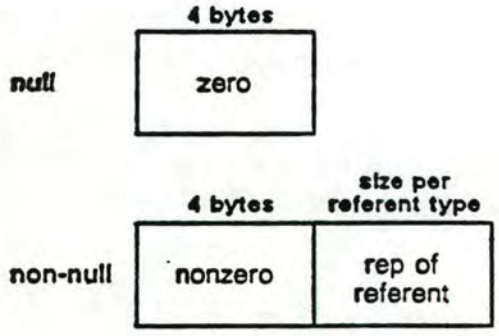

Figure 9–31. Top Level Unique Pointer Representation

9.2.11.3 Top Level Reference Pointers

A reference pointer must refer to something and cannot be null.

NDR represents a top level reference pointer simply as the representation of its referent.

Figure 9–32 illustrates a top level reference pointer as it appears in the byte stream.

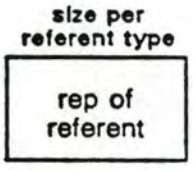

Figure 9–32. Top Level Reference Pointer Representation

9.2.12 Embedded Pointers

This subsection describes the NDR representation for pointers that are elements of arrays, members of structures, or members of unions.

In the NDR representation of an embedded pointer, the representation of the pointer referent is sometimes deferred to a later position in the byte stream while the pointer itself is represented in place as part of the constructed type. This method of representation makes possible certain optimizations in implementations of NCA.

9.2.12.1 Embedded Full Pointers

A full pointer is represented in place by an unsigned long integer. If the pointer is null, the integer has the value zero. If the pointer is non-null, the integer is the referent identifier.

The representation of the referent of a primary pointer may be deferred to a later position in the byte stream. Subsection 9.2.12.4 describes the algorithm for deferral. Except for this possible deferral, the representation of an embedded full pointer is identical to that of a top level full pointer.

Figure 9–33 illustrates the three possible representations for embedded full pointers.

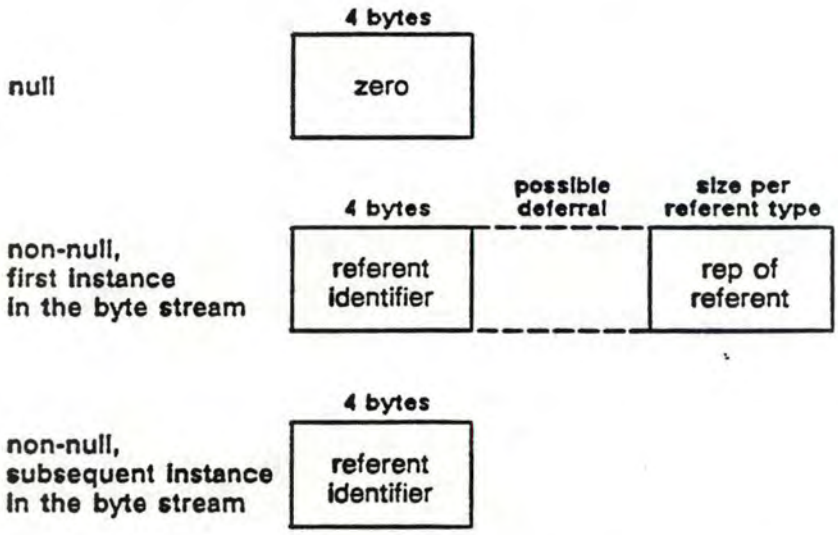

Figure 9–33. Embedded Full Pointer Representation

9.2.12.2 Embedded Unique Pointers

A unique pointer is represented in place by an unsigned long integer. If the pointer is null, the integer has the value zero. If the pointer is non-null, the integer has an arbitrary nonzero value.

The representation of the referent of a unique pointer may be deferred to a later position in the byte stream. Subsection 9.2.12.4 describes the algorithm for deferral. Except for this possible deferral, the representation of an embedded unique pointer is identical to that of a top level unique pointer.

Figure 9–34 illustrates embedded null and non-null unique pointers as they appear in the byte stream.

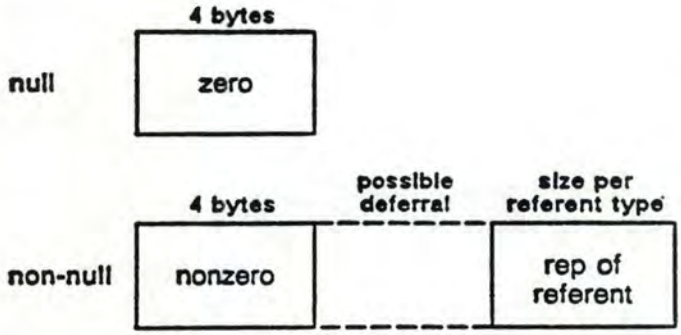

Figure 9–34. Embedded Unique Pointer Representation

9.2.12.3 Embedded Reference Pointers

An embedded reference pointer is represented in two parts, a four byte value in place and a possibly deferred representation of the referent.

The reference pointer itself is represented in place by four bytes of arbitrary value.

The representation of the referent of the reference pointer may be deferred to a later position in the byte stream. Subsection 9.2.12.4 describes the algorithm for deferral.

Figure 9–35 illustrates an embedded reference pointer as it appears in the byte stream.

Figure 9–35. Embedded Reference Pointer Representation

9.2.12.4 Algorithm for Deferral of Referents

If a pointer is embedded in an array, structure, or union, the representation of its referent is deferred to a position in the byte stream following the representation of the construction that embeds the pointer. Representations of pointer referents are ordered according to a left-to-right, depth-first traversal of the embedding construction. Following is an elaboration of the deferral algorithm in detail:

- If an array, structure, or union embeds a pointer, the representation of the referent of the pointer is deferred to a position in the byte stream following the representation of the embedding construction.
- If an array or structure embeds more than one pointer, all pointer referent representations are deferred, and the order in which referents are represented is the order in which their pointers appear in the array or structure.
- If an array, structure, or union embeds another array, structure, or union, referent representations for the embedded construction are further deferred to a position in the byte stream following the representation of the embedding construction. The set of referent representations for the embedded construction is inserted among the referent representations for any pointers in the embedding construction, according to the order of members in the embedding structure.
- The deferral of referent representations iterates through all successive embedding arrays, structures, and unions to the outermost array, structure, or union.

9.3 NDR Parameter Representations

The data type representations defined in Sections 9.1 and 9.2 are applied to the parameters transmitted as inputs and outputs in remote procedure calls.

NDR defines two modes of parameter representation: "attenuated" representation and "full" representation.

9.3.1 Attenuated Parameter Representation

In the attenuated NDR representation of a parameter, the data type representations defined in Sections 9.1 and 9.2 are applied, but some parts of those representations are omitted.

Attenuated representation can be used when a receiver does not require the actual value of a parameter but requires information about how to store the parameter. Examples of such information include the topology of a pointer-based data structure and the length of a varying array.

An attenuated representation includes only the following information:

- Union discriminators
- The "in place" representations of full pointers
- The "in place" representations of unique pointers
- Array conformance information (maximum counts)
- Array variance information (actual counts)

9.3.2 Full Parameter Representation

In the full NDR representation of a parameter, the parameter is represented according to its data type exactly as defined in Sections 9.1 and 9.2.

Full representation is used whenever a receiver requires the actual value of a parameter.

9.4 NDR Input and Output Streams

NDR represents the set of inputs or outputs in a remote procedure call as a byte stream. The byte stream consists of two parts; one part represents parameters that are pipes and the other part represents parameters that are not pipes. The first value in each part of the byte stream is aligned at a byte stream index that is a multiple of eight. To produce this alignment, a gap of bytes of arbitrary value may separate the two parts. The figures in this section do not show such gaps.

In the representation of a set of input parameters, the representations of pipes appear last. In the representation of a set of output parameters, the representations of pipes appear first.

Figure 9–36 illustrates the byte stream that represents a set of inputs. Figure 9–37 illustrates the byte stream that represents a set of outputs.

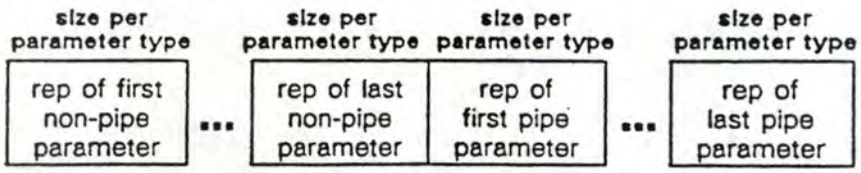

Figure 9–36. An NDR Input Stream

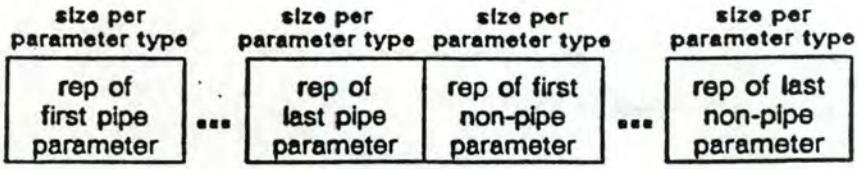

Figure 9–37. An NDR Output Stream

The byte stream representing a set of inputs or outputs is transmitted either as the body of one packet or as the bodies of several packets, as described in [].

9.5 Data Representation Format Label

The NDR format label is a vector of bytes that identifies the particular data representation formats used to represent scalar values in both the header and the body of an NCA/RPC packet. The format label is itself part of the packet header. See the NCA/RPC packet definition in [].

Figure 9–38 illustrates the NDR format label. The four most significant bits of the first byte indicate integer format. The four least significant bits of the first byte indicate character format. The second byte indicates floating-point representation format. The third and fourth bytes are reserved for future use and must contain 0s.

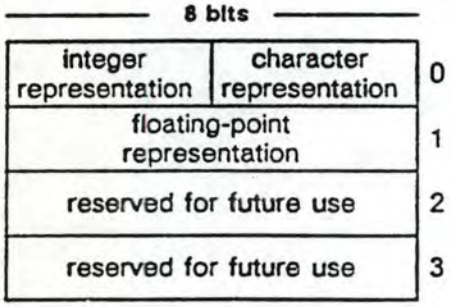

Figure 9–38. Data Representation Format Label

Table 9–3 lists the values associated with integer, character, and floating-point formats.

Table 9–3. Format Label Values

Data Type	Value in Label	Format
character	0	ASCII
	1	EBCDIC
integer	0	big-endian
	1	little–endian
floating-point	0	IEEE
	1	VAX
	2	Cray
	3	IBM

Annexe E:

Exemple BINOP

binop.idl

```
%c
[uuid(41979f30a000.0d.00.00.fb.40.00.00.00), port(dds:[19], ip:[6677]),
    version(1)]
interface binop
{
[idempotent]
void binop$add(
    handle_t [in] h,
    long [in] a,
    long [in] b,
    long [out] *c
    );
}
```

binop.h

```
#ifndef binop_included
#define binop_included
#include "idl_base.h"
#include "rpc.h"
static rpc_$if_spec_t binop$if_spec = {
    1,
    {0, 0, 6677, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    1,
    {0x41979f30,
     0xa000,
     0,
     0xd,
     {0x0, 0x0, 0xfb, 0x40, 0x0, 0x0, 0x0}}
};
extern void binop$add
#ifdef __STDC__
    (
      /* [in] */ handle_t h,
      /* [in] */ ndr_$long_int a,
      /* [in] */ ndr_$long_int b,
      /* [out] */ ndr_$long_int * c);
#else
    ();
#endif
typedef struct binop$epv_t {
    void (*binop$add)
#ifdef __STDC__
        (
          /* [in] */ handle_t h,
          /* [in] */ ndr_$long_int a,
          /* [in] */ ndr_$long_int b,
          /* [out] */ ndr_$long_int * c)
#else
        ()
#endif
        ;
} binop$epv_t;
globalref binop$epv_t binop$client_epv;
globalref binop$epv_t binop$manager_epv;
globalref rpc_$epv_t binop$server_epv;
#endif
```

binop_cswtch.c

```
#define NIDL_GENERATED
#define NIDL_CSWTCH
#include "binop.h"

void binop$add(h, a, b, c)
    handle_t h;
    ndr_$long_int a;
    ndr_$long_int b;
    ndr_$long_int *c;
{
    (*binop$client_epv.binop$add) (h, a, b, c);
}
```

binop_cstub.c

```
#define NIDL_GENERATED
#define NIDL_CSTUB
#include "binop.h"
#ifndef IDL_BASE_SUPPORTS_V1
The version of idl_base.h is not compatible with this stub / switch code.
#endif
#ifndef RPC_IDL_SUPPORTS_V1
   The version of rpc.idl is not compatible with this stub / switch code.
#endif
#ifndef NCASTAT_IDL_SUPPORTS_V1
   The version of ncastat.idl is not compatible with this stub / switch code.
#endif
#include "pfm.h"

static void binop$add_csr
#ifdef __STDC__
     (
       /* [in] */ handle_t h_,
       /* [in] */ ndr_$long_int a_,
       /* [in] */ ndr_$long_int b_,
       /* [out] */ ndr_$long_int * c_)
#else
     (h_, a_, b_, c_)
#endif
#ifndef __STDC__

/* parameter declarations */
    handle_t h_;
    ndr_$long_int a_;
    ndr_$long_int b_;
    ndr_$long_int *c_;

#endif
{
```

binop_cstub.c, cont'd.

```
/* rpc_$sar arguments */
volatile rpc_$ppkt_t *ip;
ndr_$ulong_int ilen;
rpc_$ppkt_t *op;
rpc_$ppkt_t *routs;
ndr_$ulong_int olen;
rpc_$drep_t drep;
ndr_$boolean free_outs;
status_$t st;

/* other client side local variables */
rpc_$ppkt_t ins;
rpc_$ppkt_t outs;
pfm_$cleanup_rec cleanup_rec;
status_$t cleanup_status;
ndr_$ushort_int data_offset;
ndr_$ulong_int bound;
rpc_$mp_t mp;
rpc_$mp_t dbp;
ndr_$ushort_int count;
volatile ndr_$boolean free_ins;

cleanup_status = pfm_cleanup(cleanup_rec);
if (cleanup_status.all != pfm_$cleanup_set) {
    if (free_ins)
        rpc_$free_pkt(ip);
    pfm_signal(cleanup_status);
}

/* marshalling init */
data_offset = h_->data_offset;
bound = 0;

/* bound calculations */
bound += 8;

/* buffer allocation */
if (free_ins = (bound + data_offset > sizeof(rpc_$ppkt_t)))
    ip = rpc_$alloc_pkt(bound);
else
    ip = &ins;
rpc_$init_mp(mp, dbp, ip, data_offset);
```

binop_cstub.c, cont'd.

```
    /* marshalling */
    rpc_$marshall_long_int(mp, a_);
    rpc_$advance_mp(mp, 4);
    rpc_$marshall_long_int(mp, b_);
    rpc_$advance_mp(mp, 4);

    /* runtime call */
    ilen = mp - dbp;
    op = &outs;
    rpc_$sar(h_,
             (long) 0 + rpc_$idempotent,
             &binop$if_spec,
             0L,
             ip,
             ilen,
             op,
             (long) sizeof(rpc_$ppkt_t),
             &routs,
             &olen,
             (rpc_$drep_t *) & drep,
             &free_outs,
             &st);

    /* unmarshalling init */
    rpc_$init_mp(mp, dbp, routs, data_offset);
    if (rpc_$equal_drep(drep, rpc_$local_drep)) {

        /* unmarshalling */
        rpc_$unmarshall_long_int(mp, (*c_));
    }
    else {
        rpc_$convert_long_int(drep, rpc_$local_drep, mp, (*c_));
    }

    /* buffer deallocation */
    if (free_outs)
        rpc_$free_pkt(routs);
    if (free_ins)
        rpc_$free_pkt(ip);
    pfm_rls_cleanup(cleanup_rec, cleanup_status);

}
globaldef binop$epv_t binop$client_epv = {
    binop$add_csr
};
```

binop_sstub.c

```
#define NIDL_GENERATED
#define NIDL_SSTUB
#include "binop.h"
#ifndef IDL_BASE_SUPPORTS_V1
The version of idl_base.h is not compatible with this stub / switch code.
#endif
#ifndef RPC_IDL_SUPPORTS_V1
   The version of rpc.idl is not compatible with this stub / switch code.
#endif
#ifndef NCASTAT_IDL_SUPPORTS_V1
   The version of ncastat.idl is not compatible with this stub / switch code.
#endif
#include "pfm.h"

static void binop$add_ssr
#ifdef __STDC__
    (
        handle_t h,
        rpc_$ppkt_t * ins,
        ndr_$ulong_int ilen,
        rpc_$ppkt_t * outs,
        ndr_$ulong_int omax,
        rpc_$drep_t drep,
        rpc_$ppkt_t ** routs,
        ndr_$ulong_int * olen,
        ndr_$boolean * free_outs,
        status_$t * st
)
#else
    (
        h,
        ins, ilen,
        outs, omax,
        drep,
        routs, olen,
        free_outs,
        st)
    handle_t h;
    rpc_$ppkt_t *ins;
    ndr_$ulong_int ilen;
    rpc_$ppkt_t *outs;
    ndr_$ulong_int omax;
    rpc_$drep_t drep;
    rpc_$ppkt_t **routs;
    ndr_$ulong_int *olen;
    ndr_$boolean *free_outs;
    status_$t *st;
#endif
```

binop_sstub.c, cont'd.

```
{
    /* marshalling variables */
    ndr_$ushort_int data_offset;
    ndr_$ulong_int bound;
    rpc_$mp_t mp;
    rpc_$mp_t dbp;
    ndr_$ushort_int count;

    /* local variables */
    ndr_$long_int c_;
    ndr_$long_int b_;
    ndr_$long_int a_;

    /* unmarshalling init */
    data_offset = h->data_offset;
    rpc_$init_mp(mp, dbp, ins, data_offset);
    if (rpc_$equal_drep(drep, rpc_$local_drep)) {

        /* unmarshalling */
        rpc_$unmarshall_long_int(mp, a_);
        rpc_$advance_mp(mp, 4);
        rpc_$unmarshall_long_int(mp, b_);
    }
    else {
        rpc_$convert_long_int(drep, rpc_$local_drep, mp, a_);
        rpc_$advance_mp(mp, 4);
        rpc_$convert_long_int(drep, rpc_$local_drep, mp, b_);
    }

    /* server call */
    binop$add(h, a_, b_, &c_);
    bound = 0;

    /* bound calculations */
    bound += 4;

    /* buffer allocation */
    if (*free_outs = (bound > omax))
        *routs = rpc_$alloc_pkt(bound);
    else
        *routs = outs;
    rpc_$init_mp(mp, dbp, *routs, data_offset);

    /* marshalling */
    rpc_$marshall_long_int(mp, c_);
    rpc_$advance_mp(mp, 4);

    *olen = mp - dbp;

    st->all = status_$ok;
}
```

Annexe F :

Etudes et mesures

Annexe F.1 :

Fiabilité

4. Dependability

What are the chances an RPC call will be silently ignored or executed more than once?*

If an RPC system is implemented on top of TCP, then its dependability is guaranteed by TCP. Netwise and Sun (tcp) were therefore assumed dependable, and were not tested. If an RPC system is implemented on top of UDP, however, then the RPC system must guarantee each server call is executed exactly once, since UDP packets can be lost or duplicated. In order to test dependability a simple client was written which sent an integer to the

---

* Note: this is not a measure of RPC errors, but rather errors which are not reported to the program issuing the RPC.

server, and then incremented it. The server compared the integer sent by the client to its own counter, printing a message if they were different, and incrementing its internal counter. Losing or duplicating an RPC call would cause the counters to become out of sync, triggering an error message from the server.

This program was run on the same network on which the speed tests were run under, except only the server was a 3/140. The clients were either 3/140s or 3/60s with 12 megabytes of RAM. The load on the server machine was artificially raised. This was done by running small programs which were infinite loops and had no disk operations.

Dependability
(errors/1,000,000 calls)

server load average:	~0.1	~5.0	~10.0	~20.0
Apollo	0	0	0	†
Sun (udp)	0	0*	10*	300

4.1. Discussion

The major danger is that a server routine will be executed twice. This can happen in two situations. If the data returning from the server to the client is lost (which can happen with UDP), the client will time out and resend the request, causing the server to execute its routine twice. Or, the initial data from the client to the server could arrive twice (this can also happen with UDP), also causing the server to execute the service routine twice.

To see how dependability can affect an application, consider a simple banking system. Assume the average user will have 10 bank transactions a week, and each transaction requires 20 RPC calls. For this application an RPC system which fails every million calls will cause one failure per customer per 95 years, which sounds inconsequential. But, if the bank has five million customers, that becomes over fifty thousand errors a year. Another example would be an engineering application which makes 1000 RPC calls a day and is used every working day (250 in a year). A failure rate of one in a million translates to one

† Apollo's RPC can not operate at this load. It fails, causing a core dump after 2,000 to 30,000 RPC calls.

* At one point there was a network error while running the test suites and all of the servers died. (Both Sun and Apollo with loads of 5, 10, and 20 were running at the time.) The Apollo servers died "clean" but the Sun servers returned 3 or 4 errors. These were not counted.

failure every four years.

With Sun (udp) it is important to realize that these silent errors can occur and take appropriate action.

(1) Switch to a dependable RPC system.

(2) Decide the rate of errors is acceptable and does not need to be fixed.

(3) Have the server routine check for duplicates (add an ID number to each RPC request).

(4) Define the server routine(s) to be idempotent†.

An example of ''tuning'' an application to be more dependable is Sun's Network File System (NFS). Obviously, a file system must be very reliable, even under overloaded conditions, or its users will revolt. Sun achieved this while using their UDP based RPC by making most NFS server routines idempotent, and having the server check the rest to make sure they could not be executed twice.

---

† An idempotent routine can be executed more than once without disturbing the system Examples: gethostname, fseek. Counter Examples: unlink, lockf.

Annexe F.2 :

Performance

RPC Timing Performance
The Raw Speed Numbers

location of server and client(s):			same machine	different machine
Implementation	Clients	Size	Speed	Speed
Sun (udp)	1	12	4.2	4.3
Sun (tcp)	1	12	5.6	6.0
Apollo	1	12	4.0	4.4
Netwise	1	12	11.1	11.9
Sun (udp)	2	12	6.4	6.0
Sun (tcp)	2	12	11.7	9.0
Apollo	2	12	7.2	7.5
Netwise	2	12	15.7	15.3
Sun (udp)	4	12	13.9	12.3
Sun (tcp)	4	12	22.6	17.3
Apollo	4	12	14.1	12.1
Netwise	4	12	30.1	25.7
Sun (udp)	1	512	4.3	4.7
Sun (tcp)	1	512	6.0	6.7
Apollo	1	512	5.3	5.7
Netwise	1	512	13.4	14.2
Sun (udp)	2	512	7.9	6.4
Sun (tcp)	2	512	15.8	10.3
Apollo	2	512	11.0	8.0
Netwise	2	512	20.2	19.3
Sun (udp)	4	512	14.0	12.5
Sun (tcp)	4	512	27.4	19.9
Apollo	4	512	19.3	15.9
Netwise	4	512	38.0	53.4
Sun (udp)	1	8K	5.9	9.6
Sun (tcp)	1	8K	11.4	13.4
Apollo	1	8K	42.0	43.2
Sun (udp)	2	8K	16.8	23.2
Sun (tcp)	2	8K	23.1	23.4
Apollo	2	8K	81.6	70.8
Sun (udp)*	4	8K	38.0	34.6
Sun (tcp)	4	8K	46.3	42.0
Apollo	4	8K	152.9	126.1
Sun (tcp)	1	16K	18.2	21.8
Apollo	1	16K	91.5	84.0
Sun (tcp)	2	16K	34.2	34.4
Apollo	2	16K	161.4	147.2
Sun (tcp)	4	16K	68.2	69.2
Apollo	4	16K	298.6	248.5

Clients is the number of processes accessing the server at once.
Size is the packet size, in bytes.

* One timeout error occurred during this test.

Comparative Commercial RPCs

Speed is seconds, rounded to the nearest tenth, for 100 remote procedur

All tests run on Sun 3/140s with 8 Megs of memory on a lightly loaded e

- 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

Remote Procedure Call Performance

A Review of Current Product Offerings

Introduction

This paper presents preliminary results of an ongoing performance study of the major commercial remote procedure call (RPC) products. The testing thus far has measured round-trip times for remote procedure calls using a variety of simple data types, as well as the basic code size overhead for a simple remote procedure call.

The products used for this comparison are:

- Netwise C Language RPC TOOL, version 2.02
- Apollo NCS, version 1.1
- Sun/RPC, version 3.9

Testing Environment

All tests were performed on a Sun 3/280 system running SunOS 3.5 in single-user mode. Client and server processes executed within the same system. Each test run measured the elapsed real time for 500 calls. The times shown represent the average of five such test runs.

We attempted to measure only the remote procedure call overhead itself. In each case, client/server binding overhead was minimized by using the most optimal binding method and excluding any set-up calls from the measured time. In addition, all procedures were declared as "idempotent" for the Apollo tests.

The tests used are as follows:

void	void procedure with no parameters
add	adds two integers and returns the result
integer 100	integer array of 100 elements as the sole input parameter; returns an integer
integer 1000	same as above but with 1000 elements
string	concatenates two null-terminated string parameters and returns the result; input strings are 15 and 18 characters long

opaque 4000	passes and returns a buffer of 4000 bytes; also returns an integer
opaque 8000	passes and returns a buffer of 8000 bytes; also returns an integer
opaque 16000	passes and returns a buffer of 16000 bytes; also returns an integer

The tests were implemented as uniformly as possible across the three products tested. In some cases an output parameter was used instead of a return value due to RPC compiler restrictions, but no significant changes were required for any of the RPC compilers.

Testing was done for both TCP and UDP transports wherever possible.

Timing Results

The following table shows the measured round-trip per-call times (in milliseconds) as measured in the client process, excluding extraneous binding and set-up overhead.

	TCP Transport Round-Trip Time (ms)			UDP Transport Round-Trip Time (ms)		
	Sun	Apollo	Netwise	Sun	Apollo	Netwise
Void	5.01	-	5.34	4.11	5.95	4.37
Add	5.22	-	5.47	4.22	6.82	4.48
Integer 100	10.37	-	11.42	8.60	10.38	9.99
Integer 1000	57.03	-	65.35	46.41	81.04	59.01
String	6.38	-	7.11	5.17	11.49	6.00
Opaque 4000	28.30	-	72.38	15.38	131.66	61.15
Opaque 8000	62.05	-	140.40	25.09	353.06	116.82
Opaque 16000	105.27	-	272.05	-	431.31	-

Some of the RPC times above warrant further explanation. No TCP times are presented for Apollo because NCS supports only datagram transports such as UDP. Times are also missing for Netwise and Sun for UDP in the "opaque 16000" test case. This is because of the size limit for UDP messages. Apollo has avoided this limit through additional protocols implemented above UDP.

The Netwise UDP times were measured using a pre-release product. All other times represent currently available products.

Annexe F.3 :

Taille des exécutables

The slower performance of Netwise with respect to Sun in the opaque tests is due to the use of a suboptimal ASN.1 constructed encoding. Future releases will use a simpler ASN.1 encoding, bringing the numbers more in line with Sun.

Code Size Results

Code sizes for a minimal distributed application (the void test from above) were measured. The sizes were broken down into the following categories:

- Application Code
  User-written application code. On the client side, this includes any binding calls required by the RPC system.
- Server Initialization
  This is the server program's main routine, which generally performs some network initialization and begins waiting for RPC requests. In the case of Apollo's NCS, this code is written by the user.
- Generated Code
  Code generated by the RPC compiler for client/server stubs, data transfer procedures, and so forth.
- Libraries
  Code obtained by linking with runtime libraries provided by the RPC vendor. This does not include any code from the standard C runtime library.

The observed code sizes are as follows:

	Client Object Code Size (bytes)			Server Object Code Size (bytes)		
Code Type	Sun	Apollo	Netwise	Sun	Apollo	Netwise
Application	304	452	264	16	20	20
Server Init	-	-	-	300	356	1108
Gen'd Code	100	132	132	276	52	984
Libraries	12216	38836	9964	17884	38836	9396
TOTAL	12620	39420	10360	18476	39264	11508

The slower performance of Netwise with respect to Sun in the opaque tests is due to the use of a suboptimal ASN.1 constructed encoding. Future releases will use a simpler ASN.1 encoding, bringing the numbers more in line with Sun.

Code Size Results

Code sizes for a minimal distributed application (the void test from above) were measured. The sizes were broken down into the following categories:

- Application Code
  User-written application code. On the client side, this includes any binding calls required by the RPC system.
- Server Initialization
  This is the server program's main routine, which generally performs some network initialization and begins waiting for RPC requests. In the case of Apollo's NCS, this code is written by the user.
- Generated Code
  Code generated by the RPC compiler for client/server stubs, data transfer procedures, and so forth.
- Libraries
  Code obtained by linking with runtime libraries provided by the RPC vendor. This does not include any code from the standard C runtime library.

The observed code sizes are as follows:

	Client Object Code Size (bytes)			Server Object Code Size (bytes)		
Code Type	Sun	Apollo	Netwise	Sun	Apollo	Netwise
Application	304	452	264	16	20	20
Server Init	-	-	-	300	356	1108
Gen'd Code	100	132	132	276	52	984
Libraries	12216	38836	9964	17884	38836	9396
TOTAL	12620	39420	10360	18476	39264	11508

Annexe G :

Protocole NCA/RPC

Chapter 5

NCA/RPC Finite State Machine Definitions, Notations, and Conventions

The NCA/RPC request-response protocol is divided into client and server sides. This specification defines the client and server as finite state machines (FSMs). The client consists of four FSMs, while the server consists of one FSM. The client and server FSMs are designed to operate together: the output of the client FSMs is input to the server FSM, and the output of the server FSM is input to the client FSMs.

Chapters [] and [] specify the client and server FSMs, respectively. This chapter defines the components of the FSMs and describes the conventions we use in state transition tables.

5.1 Definitions

An FSM defines a set of states and state transitions. An FSM has a current state which changes as a result of inputs. The next state is determined by reference to a set of possible state transitions. A state transition is a (state, input, condition, next state, action) tuple. A transition is considered possible if and only if its condition is true. All transitions for a state are ordered. If an FSM in state S receives input I, the next state is the state that results from the first possible transition of the form (S, I, ...), and the action associated with that transition is executed.

The rest of this section defines the inputs, conditions, and actions associated with state transitions.

5.1.1 Inputs

FSM inputs can be

- Messages. Message input occurs when an NCA/RPC packet arrives. The set of messages is the set of NCA/RPC packet types. We define packet types in [].
- Timeouts. Timeout inputs are associated with certain states and occur after a certain amount of time has passed in that state with no other inputs. We define the set of client timeouts in [] and the set of server timeouts in [].
- Client actions. A client action is either a request to execute a remote procedure call (*ClientCall*) or a request to terminate a call in progress (*ClientQuit*).
- Server execution engine notifications. The execution engine is the model for the execution of a remote operation. The engine executes an operation when invoked by the server FSM; it returns any results as notification input to the server FSM. Execution engine notification input indicates that the execution engine has changed its state. We list notification inputs in []. Server FSM primitives that invoke the execution engine are listed in [] and defined in [].

Messages, client actions, and server execution engine notifications have structured bundles of data associated with them. The nature of the data in a bundle depends on the type of FSM input.

5.1.1.1 Message Input Bundles

The bundle associated with message input consists of

- NCA/RPC packet header field values, as described in []. Table 5–1 lists the packet header field bundle data and the packet header fields to which they correspond.
- NCA/RPC packet flag values, as described in []. A packet flag value is a boolean expression which is true if and only if the named flag is set in the packet header. Table 5–2 lists the packet flag bundle data and the packet flags to which they correspond.
- The source address of the message sender, which is a socket address, as defined in nbase.idl and described in [] [?]. The source address bundle data identifies the location from which the message input originated.
- The NCA/RPC packet body, which contains the input or output parameters for a remote procedure call.

6.5 Client Constants

Table 6–5 describes the constants used in the client FSMs.

Table 6–5. Client Constants

Name	Type	Meaning
ActivityID	UUID	The activity ID of the caller making the request.
MaxBodyLen	integer	Maximum number of bytes in the body of a single-packet *request*.
MaxPings	integer	Maximum number of unacknowledged *pings* that should be sent in a row. Suggested value: 30.
MaxQuits	integer	Maximum number of unacknowledged *quits* that should be sent before entering the done state. Suggested value: 3.
MaxRequests	integer	Maximum number of *requests* that should be sent for a single remote call. Suggested value: 5.

6.6 Client Global Variables

Table 6–6 describes the global variables used in the client FSMs.
[Give initial value for bootTime?]

Table 6–6. Client Global Variables

Name	Type	Meaning
bootTime	integer	The client's record of the time the server last booted.
callSpec	aggregate type	A copy of the NDR representation of the call specification from the *ClientCall* [callSpec] bundle.
idempotent	Boolean	True if and only if the current call is to an idempotent operation.
inFragNum	integer	The fragment number of the last fragment of input data sent (zero-based).
inParams	array of bytes	A copy of the NDR representation of the input parameters from the *ClientCall* [inParams] input bundle.
outFragNum	integer	The fragment number of the last fragment of output data received in order (zero-based).
outParams	array of bytes	The reassembled (unfragmented) output data associated with the current request.
pingCount	integer	The number of consecutively sent unacknowledged *ping* packets.
quitCount	integer	The number of times a *quit* packet has been sent.
requestCount	integer	The number of times a *request* packet has been sent for the current call.
seqNum	integer	The sequence number of the current call. Initial value: 0.
waitCount	integer	The number of times the FSM has entered the wait state without sending a *request* packet first. This variable determines how long the FSM waits in the wait state.

6.7 Client Primitives

Table 6–7 lists the primitives used in client FSM actions. [] defines their syntax.

Table 6–7. Client Primitives

Primitive	Meaning
RaiseException	Reflect an error that occurred in the FSM to the calling client.
SendPkt	Build a packet and send it over the communications medium.
SendBroadcastPkt	Build a packet and broadcast it over the communications medium.
SendMaybePkt	Build a packet and send it over the communications medium.
SendBroadcastMaybePkt	Build a packet and broadcast it over the communications medium.

6.8 Client FSM Tables

The tables in this section define the send-await-reply, broadcast, maybe, and broadcast/maybe finite state machines.

Changes to Client SAR FSM since 1.5.1 version:

done	@FragTimeout	/	done	*send ack*
changed to				
done	@AckTimeout	/	init	*send ack*
added				
done	*ClientCall*	lengthInFrag([inParams]) > 1	fack_wait	*send frag request*

Table 6–8. Send-Await-Reply FSM

State	Input	Condition	Next State	Action
init	*ClientQuit*	/	init	/
init	*ClientCall*	lengthInFrags([inParams]) > 1	fack_wait	*send frag request*
init	*ClientCall*	/	wait	*send request*
init	*response*	/	init	*send ack*
wait	/	requestCount > MaxRequests	done	*comm failure*
wait	*ClientQuit*	/	quit_wait	*send quit*
wait	@WaitTimeout	/	ping_wait	*start pinging*
wait	*fault*	/	done	*handle error*
wait	*reject*	/	done	*handle error*
wait	*working*	/	wait	*wait longer*
wait	*nocall*	/	wait	*resend request*
wait	*response*	[frag] & [fragNum] $\neq$ outFragNum+1 & [nofack]	wait	/
wait	*response*	[frag] & [fragNum] $\neq$ outFragNum+1	wait	*frag ack*
wait	*response*	[lastFrag]	done	*handle response*
wait	*response*	[frag] & [nofack]	wait	*handle frag*
wait	*response*	[frag]	wait	*handle frag; frag ack*
wait	*response*	/	done	*handle response*
ping_wait	/	pingCount > MaxPings	done	*comm failure*
ping_wait	*ClientQuit*	/	quit_wait	*send quit*
ping_wait	*working*	/	wait	*wait longer*
ping_wait	*nocall*	/	wait	*resend request*
ping_wait	*fault*	/	done	*handle error*
ping_wait	*reject*	/	done	*handle error*

(continued)

Table 6–8. Send-Await-Reply FSM (continued)

State	Input	Condition	Next State	Action
ping_wait	*response*	[frag] & [fragNum] ≠ outFragNum+1 & [nofack]	wait	/
ping_wait	*response*	[frag] & [fragNum] ≠ outFragNum+1	wait	*frag ack*
ping_wait	*response*	[lastFrag]	done	*handle response*
ping_wait	*response*	[frag] & [nofack]	wait	*handle frag*
ping_wait	*response*	[frag]	wait	*handle frag; frag ack*
ping_wait	*response*	/	done	*handle response*
ping_wait	@PingTimeout	/	ping_wait	*send ping*
fack_wait	*ClientQuit*	/	quit_wait	*send quit*
fack_wait	*fack*	[fragNum] ≠ inFragNum	fack_wait	*wait longer*
fack_wait	*fack*	inFragNum < lengthinFrags(inParams)	fack_wait	*send next frag*
fack_wait	*fack*	/	wait	*send last frag*
fack_wait	*nocall*	/	fack_wait	*resend request*
fack_wait	@FragTimeout	/	fack_wait	*resend request*
quit_wait	/	quitCount > MaxQuits	done	/
quit_wait	@QuitTimeout	/	quit_wait	*send quit*
quit_wait	*quack*	/	done	/
quit_wait	*response*	/	done	/
quit_wait	*nocall*	/	done	/
done	/	Idempotent	init	/
done	*ClientCall*	lengthInFrags([inParams]) > 1	fack_wait	*send frag request*
done	*ClientCall*	/	wait	*send request*
done	@AckTimeout	/	init	*send ack*

Table 6-9. Send-Await-Reply FSM Actions

send frag request *setup request* *SendPkt* (*request*, [frag], idempotent, seqNum, inFragNum, callSpec, inParams[inFragNum])
send request *setup request* *SendPkt* (*request*, [], idempotent, seqNum, 0, callSpec, inParams)
setup request seqNum++ idempotent := [idempotent] callSpec := [callSpec] inParams := [inParams] requestCount := 0 inFragNum := 0 outFragNum := −1 waitCount := 0 quitCount := 0 requestCount := 0 outParams := NULL
send ack *SendPkt* (*ack*, [], false, seqNum, 0, callSpec, NULL)
comm failure *RaiseException* (CommFailure)
send quit quitCount++ *SendPkt* (*quit*, [], false, seqNum, 0, callSpec, NULL)
start pinging pingCount := 0 *send ping*
send ping pingCount++ *SendPkt* (*ping*, [], false, seqNum, 0, callSpec, NULL)

(continued)

Table 6–9. Send-Await-Reply FSM Actions (continued)

handle error *RaiseException* (NDRtoStatus([body]))
wait longer waitCount++
resend request requestCount++ waitCount := 0 *SendPkt* (*request*, [], idempotent, seqNum, inFragNum, callSpec, inParams[inFragNum])
frag ack *SendPkt* (*fack*, [], false, seqNum, outFragNum, callSpec, NULL)
handle response outParams := outParams ⊕ [body] bootTime := [bootTime]
handle frag outParams := outParams ⊕ [body] outFragNum := [fragNum]
send next frag requestCount := 0 waitCount := 0 inFragNum++ *SendPkt* (*request*, [frag], idempotent, seqNum, inFragNum, callSpec, inParams[inFragNum])
send last frag requestCount := 0 waitCount := 0 inFragNum++ *SendPkt* (*request*, [frag, lastFrag], idempotent, seqNum, inFragNum, callSpec, inParams[inFragNum])

Table 6-10. Broadcast FSM

State	Input	Condition	Next State	Action
init	*ClientCall*	/	**wait**	*send request*
wait	@BrdcstTimeout	/	**done**	*comm failure*
wait	*fault*	/	**done**	*handle error*
wait	*reject*	/	**done**	*handle error*
wait	*response*	/	**done**	*handle response*
done	*ClientCall*	/	**wait**	*send request*
done	/	/	**init**	/

Table 6-11. Broadcast FSM Actions

send request *setup request* *SendBroadcastPkt* ([], seqNum, 0, callSpec, inParams)
setup request seqNum++ broadcast := [broadcast] callSpec := [callSpec] inParams := [inParams] requestCount := 0 outParams := NULL
comm failure *RaiseException* (CommFailure)
handle error *RaiseException* (NDRtoStatus([body]))
handle response outParams := outParams ⊕ [body] bootTime := [bootTime]

Table 6–12. Maybe FSM

State	Input	Condition	Next State	Action
init	*ClientCall*	/	**done**	*send request*
done	*ClientCall*	/	**done**	*send request*
done	/	/	**init**	/

Table 6–13. Maybe FSM Actions

send request
setup request
SendMaybePkt ([], seqNum, 0, callSpec, inParams)

setup request
seqNum++
maybe := [maybe]
callSpec := [callSpec]
inParams := [inParams]
outParams := NULL

Table 6–14. Broadcast/Maybe FSM

State	Input	Condition	Next State	Action
init	*ClientCall*	/	**done**	*send request*
done	*ClientCall*	/	**done**	*send request*
done	/	/	**init**	/

Table 6–15. Broadcast/Maybe FSM Actions

send request
setup request
SendBroadcastMaybePkt ([], seqNum, 0, callSpec, inParams)

setup request
seqNum++
broadcast := [broadcast]
maybe := [maybe]
callSpec := [callSpec]
inParams := [inParams]
outParams := NULL

6.9 Client Primitive Syntax Description

This section gives the syntax for the client FSM primitives.

NAME

RaiseException — Reflect an error to the client that is using the FSM.

SYNTAX

RaiseException (status)

INPUT PARAMETER

status A reject status code (from []).

DESCRIPTION

RaiseException reflects an error that occurred in the FSM to the client that is using the FSM, following the error signaling conventions that the client employs.

NAME

SendBroadcastPkt — Build an NCA/RPC packet and broadcast it over the communications medium.

SYNTAX

SendBroadcastPkt (pktflags, seqnum, fragnum, callspec, data)

INPUT PARAMETERS

pktflags One or more flags (from []). Determines the packet flags field in the packet header. The [broadcast] flag is always set in broadcast *request* packets.

seqnum An integer. Determines the sequence number field in the packet header.

fragnum An integer. Determines the fragment number field in the packet header.

callspec A call specification defined in the *ClientCall* input bundle [callSpec] that identifies the interface specification, object UUID, and destination of the remote procedure call. [] defines the values passed in [callSpec]. The values in [callSpec] determine the following packet header fields:

[callSpec]	Packet Header Fields
[callSpec.if_id]	if_id
[callSpec.if_vers]	if_vers
[callSpec.opnum]	opnum
[callSpec.object]	object

SendBroadcastPkt passes the values in *callspec* through to the destination without interpretation.

data A variable-length string of bytes. Determines the body portion of the packet. The length of *data* determines the body length field in the packet header.

DESCRIPTION

SendBroadcastPkt builds the packet header and body with values specified in its input parameters and broadcasts the packet over the communications medium to the target destinations specified in [callSpec]. *SendBroadcastPkt* determines the destinations from [callSpec] in the same manner as the *SendPkt* primitive. [Elaborate?]

NAME

SendBroadcastMaybePkt — Build an NCA/RPC packet and broadcast it over the communications medium.

SYNTAX

SendBroadcastMaybePkt (pktflags, seqnum, fragnum, callspec, data)

INPUT PARAMETERS

pktflags One or more flags (from []). Determines the packet flags field in the packet header. The [broadcast] and [maybe] flags are always set in broadcast/maybe *request* packets.

seqnum An integer. Determines the sequence number field in the packet header.

fragnum An integer. Determines the fragment number field in the packet header.

callspec A call specification defined in the *ClientCall* input bundle [callSpec] that identifies the interface specification, object UUID, and destination of the remote procedure call. [] defines the values passed in [callSpec]. The values in [callSpec] determine the following packet header fields:

[callSpec]	Packet Header Fields
[callSpec.if_id]	if_id
[callSpec.if_vers]	if_vers
[callSpec.opnum]	opnum
[callSpec.object]	object

SendBroadcastMaybePkt passes the values in *callspec* through to the destination without interpretation.

data A variable-length string of bytes. Determines the body portion of the packet. The length of *data* determines the body length field in the packet header.

DESCRIPTION

SendBroadcastMaybePkt builds the packet header and body with values specified in its input parameters and broadcasts the packet over the communications medium to the target destinations specified in [callSpec]. *SendBroadcastMaybePkt* determines the destinations from [callSpec] in the same manner as the *SendPkt* primitive. [Elaborate?]

NAME

SendMaybePkt — Build an NCA/RPC packet and send it over the communications medium.

SYNTAX

SendMaybePkt (pktflags, seqnum, fragnum, callspec, data)

INPUT PARAMETERS

pktflags One or more flags (from []). Determines the packet flags field in the packet header. The [maybe] flag is always set in maybe packets.

seqnum An integer. Determines the sequence number field in the packet header.

fragnum An integer. Determines the fragment number field in the packet header.

callspec A call specification defined in the *ClientCall* input bundle [callSpec] that identifies the interface specification, object UUID, and destination of the remote procedure call. [] defines the values passed in [callSpec]. The values in [callSpec] determine the following packet header fields:

[callSpec]	Packet Header Fields
[callSpec.if_id]	if_id
[callSpec.if_vers]	if_vers
[callSpec.opnum]	opnum
[callSpec.object]	object

SendMaybePkt passes the values in *callspec* through to the destination without interpretation.

data A variable-length string of bytes. Determines the body portion of the packet. The length of *data* determines the body length field in the packet header.

DESCRIPTION

SendMaybePkt builds the packet header and body with values specified in its input parameters and sends the packet over the communications medium to the target destination specified in [callSpec]. *SendMaybePkt* determines the destination from [callSpec] in the same manner as the *SendPkt* primitive.

NAME

SendPkt — Build an NCA/RPC packet and send it over the communications medium.

SYNTAX

SendPkt (pkttype, pktflags, idempotent, seqnum, fragnum, callspec, data)

INPUT PARAMETERS

pkttype A client-initiated packet type (from []). Determines the packet type field in the packet header.

pktflags One or more flags (from []). Determines the packet flags field in the packet header.

idempotent A Boolean. Determines whether the [idempotent] flag is set in the packet header.

seqnum An integer. Determines the sequence number field in the packet header.

fragnum An integer. Determines the fragment number field in the packet header.

callspec A call specification defined in the *ClientCall* input bundle [callSpec] that identifies the interface specification, object UUID, and destination of the remote procedure call. [] defines the values passed in [callSpec]. The values in [callSpec] determine the following packet header fields:

[callSpec]	Packet Header Fields
[callSpec.if_id]	if_id
[callSpec.if_vers]	if_vers
[callSpec.opnum]	opnum
[callSpec.object]	object

SendPkt passes the values in *callspec* through to the destination without interpretation.

data A variable-length string of bytes. Determines the body portion of the packet. The length of *data* determines the body length field in the packet header.

DESCRIPTION

SendPkt builds the packet header and body with values specified in its input parameters and sends the packet over the communications medium to the target destination specified in [callSpec].

[Better story on forwarding?]

SendPkt determines the destination from [callSpec] as follows:

1. If [callSpec.location] specifies a port, *SendPkt* sends the packet to that port.
2. If [callSpec.location] specifies a port of "unspecified", *SendPkt* uses the address family specified in [callSpec.location] as an index into the port vector [callSpec.port_list] and sends the packet to the indexed port.
3. If the port in [callSpec.port_list] is "unspecified", *SendPkt* uses a forwarding port vector (for example, the LLB forwarding port specified in the port attribute of the Local Location Broker (LLB) interface definition header) and uses the address family specified in [callSpec.location] as an index into that list.

Note that *SendPkt* only needs to use a forwarding port on the first remote call. On subsequent calls to the same destination, the client will use the port sent in the reply packet it received from the server as a *response* to the initial *request*.

Annexe G.2 :

FSM serveur

Chapter 7

NCA/RPC Server Protocol Specification

The NCA/RPC server protocol consists of one finite state machine that characterizes the server's response to a single client. The server FSM handles one client activity at a time. It does not specify how a particular client is selected; the mechanism used is implementation-dependent. Implementations of the NCA/RPC server protocol can be written to allow the server to handle multiple clients simultaneously; that is, they can choose to implement multiple instances of the server FSM's single-client handling in the same process. This chapter defines the timeouts, input data, constants, global variables, and subroutine primitives used in the server FSM table.

7.1 Server Timeouts

Table 7–1 defines the timeouts that the server FSM uses.

Table 7–1. Server Timeouts

Timeout	State	Meaning
@IdleTimeout	final	How long the server will hold information about the client activity (the sequence number, the activity UUID, and so on). Suggested time: 5 minutes.
@ResendTimeout	replied replying	How long the server will wait for acknowledgement from the client before retransmitting a response. Suggested time: 3 seconds.

7.2 Execution Engine Notifications

Table 7–2 defines the notifications that the execution engine returns to the server FSM.

Table 7–2. Execution Engine Notification

Event	Meaning
CallbkCompletes	The callback that the execution engine initiated in response to the *StartCallback* primitive has completed. The bundle returned with this notification is [seqNum].
InvocationError	The execution engine cannot execute the operation that the *StartApplicationProcedure* primitive requested. The bundle returned with this notification is [rejectStatus].
ProcCompletes	The execution engine has executed the operation that the *StartApplicationProcedure* primitive requested. The bundle returned with this notification is [outParams].
ProcFaults	The execution engine has executed the operation that the *StartApplicationProcedure* primitive requested. The bundle returned with this notification is [faultStatus].

7.3 Execution Engine Notification Input Data

Table 7–3 defines the bundle data associated with execution engine notification input.

Table 7–3. Execution Engine Notification Input Bundles

Name	Type	Meaning
[faultStatus]	integer	A 32-bit fault status associated with the *ProcFaults* notification.
[outParams]	array of bytes	The NDR representation of the output parameters associated with the *ProcCompletes* notification.
[rejectStatus]	integer	A 32-bit reject status associated with the *InvocationError* notification. Reject status codes are defined in [].
[seqNum]	integer	A 32-bit sequence number associated with the *CallbkCompletes* notification. The sequence number that represents the client's record of the current remote call.

7.4 Server Constants

Table 7–4 defines the constants used in the server FSM and actions tables.

Table 7–4. Constants

Name	Type	Meaning
BootTime	integer	A 32-bit value that indicates the time the server last booted, in UNIX time format (time in seconds since 1 January 1970).
MaxBodyLen	integer	Maximum length allowed for a single-packet *response*.
MaxReplies	integer	Maximum number of times a *response* should be sent.
WrongBootTime	integer	A 32-bit value that indicates that the client's record of the server's boot time is incorrect. [A status code?]

7.5 Server Global Variables

Table 7–5 defines the global variables used in server FSM and actions tables.

Table 7–5. Global Variables

Name	Type	Meaning
activityID	UUID	The universal unique identifier of the client activity that is making the remote call. This value is initially 0.
broadcast	Boolean	True only if the current call was broadcast.
fragNum	integer	The fragment number received in the packet header.
idempotent	Boolean	True only if the current call is to an idempotent procedure.
inFragNum	integer	The number of the last fragment of inParams received in order (zero-based).
inParams	array of bytes	The reassembled (unfragmented) input data associated with the current *request*.
isFrag	Boolean	True only if the frag bit in the packet flags field is set in the current *request* packet.
maybe	Boolean	True only if the current call is to a maybe procedure.
noFack	Boolean	True only if the nofack bit in the packet flags field is set in the current *request* packet.
outFragNum	integer	The number of the last fragment of output data sent (zero-based).
outParams	array of bytes	The unfragmented output data associated with the current *response*.
replyCount	integer	The number of times a reply packet has been sent for the current call.
replyType	integer	The packet type of the packet being returned to the client. Packet types are defined in [].
seqNum	integer	The sequence number of the call that the server is currently processing for the client. This value is initially –1.
sourceAddress	integer	The location of the packet's sender.

7.6 Server Primitives

Table 7–6 lists the primitives used in server FSM actions. [] defines their syntax.

Table 7–6. Server Primitives

Primitive	Meaning
KillApplicationProcedure	Direct the execution engine to stop executing a remote operation.
SendPkt	Build a packet and send it over the communications medium.
StartApplicationProcedure	Direct the execution engine to execute a remote operation and to return any output data from that operation.
StartCallback	Direct the execution engine to make a remote procedure call to the Conversation Manager running on the host from which a client *request* originated.

7.7 Server FSM Tables

The tables in this section define the server FSM.

Changes to Server FSM since 1.5.1 version:

replying	*ping*	[seqNum] = seqNum	replying	*send nocall*
changed to				
replying	*ping*	[seqNum] > seqNum	replying	*send nocall*

Table 7–7. Server FSM

State	Input	Condition	Next State	Action
init	*request*	[bootTime] ≠ BootTime	init	*send boot time error*
init	*request*	[broadcast]	working	*do request*
init	*request*	[maybe]	working	*do request*
init	*request*	[frag] & [fragNum] ≠ 0	init	/
init	*request*	[frag] & [idempotent] & [nofack]	frag	*do first Infrag*
init	*request*	[frag] & [idempotent]	frag	*do first Infrag; frag ack*
init	*request*	[idempotent]	working	*do request*
init	*request*	/	callback	*do callback*
callback	*quit*	/	init	*send quack*
callback	*CallbkCompletes*	[seqNum] ≠ seqNum	init	/
callback	*CallbkCompletes*	isFrag & noFack	frag	*do first Infrag*
callback	*CallbkCompletes*	isFrag	frag	*do first Infrag; frag ack*
callback	*CallbkCompletes*	/	working	*do request*
frag	*quit*	[seqNum] ≠ seqNum	frag	/
frag	*quit*	/	final	*send quack*
frag	*request*	[frag] & [seqNum] ≠ seqNum	frag	/
frag	*request*	[frag] & [fragNum] ≠ inFragNum+1	frag	/
frag	*request*	[frag] & [lastFrag]	working	*do request*
frag	*request*	[frag] & [nofack]	frag	*do next Infrag*
frag	*request*	[frag]	frag	*do next Infrag; frag ack*
working	*quit*	[seqNum] ≠ seqNum	working	/
working	*quit*	/	final	*quit call; send quack*
working	*ping*	[bootTime] ≠ BootTime	done	*handle error*
working	*ping*	[seqNum] > seqNum	working	*send nocall*

(continued)

Table 7-7. Server FSM (continued)

State	Input	Condition	Next State	Action
working	*ping*	/	**working**	*send working*
working	*InvocationError*	broadcast	**init**	*free client*
working	*InvocationError*	idempotent	**final**	*send reject; free reply*
working	*InvocationError*	/	**replied**	*send reject*
working	*ProcFaults*	idempotent	**final**	
working	*ProcFaults*	/	**replied**	*send fault*
working	*ProcCompletes*	maybe	**init**	*free client*
working	*ProcCompletes*	lengthInFrags([outParams]) > 1	**replying**	*send first outfrag*
working	*ProcCompletes*	broadcast	**final**	*send reply*
working	*ProcCompletes*	idempotent	**final**	*send reply*
working	*ProcCompletes*	/	**replied**	*send reply*
replying	/	replyCount > MaxReplies	**final**	*free reply*
replying	@ResendTimeout	/	**replying**	*resend outfrag*
replying	*quit*	[seqNum] = seqNum	**final**	*free reply*
replying	*quit*	/	**final**	/
replying	*ping*	[seqNum] > seqNum	**replying**	*send nocall*
replying	*ping*	/	**replying**	*resend outfrag*
replying	*fack*	[seqNum] ≠ seqNum	**replying**	/
replying	*fack*	[fragNum] ≠ outFragNum	**replying**	*resend outfrag*
replying	*fack*	outFragNum > lengthInFrags([outParams])	**replying**	*send next outfrag*
replying	*fack*	/	**replied**	*send last outfrag*
replied	/	replyCount > MaxReplies	**final**	*free reply*
replied	@ResendTimeout	/	**replied**	*resend reply*
replied	*quit*	[seqNum] ≠ seqNum	**final**	/

(continued)

Table 7-7. Server FSM (continued)

State	Input	Condition	Next State	Action
replied	*quit*	/	**final**	*free reply*
replied	*request*	[seqNum] ≠ seqNum	**replied**	/
replied	*request*	/	**replied**	/
replied	*ack*	[seqNum] ≠ seqNum	**replied**	/
replied	*ack*	/	**final**	*free reply*
replied	*ping*	[seqNum] > seqNum	**replied**	*send nocall*
replied	*ping*	/	**replied**	*resend reply*
final	@IdleTimeout	/	**init**	*free client*
final	*ping*	/	**final**	*send nocall*
final	*request*	[seqNum] <= seqNum	**final**	*send nocall*
final	*request*	[bootTime] ≠ BootTime	**final**	*send boot time error*
final	*request*	[broadcast]	**working**	*do request*
final	*request*	[maybe]	**working**	*do request*
final	*request*	[frag] & [fragNum] ≠ 0	**final**	/
final	*request*	[frag] & [nofack] & [idempotent]	**frag**	*do first Infrag*
final	*request*	[frag] & [idempotent]	**frag**	*do first Infrag; frag ack*
final	*request*	[idempotent]	**working**	*do request*
final	*request*	/	**callback**	*do callback*

Table 7–8. Server FSM Actions

send boot time error *SendPkt* (*reject*, [], seqNum, 0, StatustoNDR(WrongBootTime), sourceAddress)
do request sourceAddress := [sourceAddress] outParams := NULL outFragNum := 0 inParams := inParams ⊕ [body] idempotent := [idempotent] maybe := [maybe] broadcast := [broadcast] seqNum := [seqNum] ifID := [ifID] ifVers := [ifVers] objectID := [objectID] opNum := [opNum] *StartApplicationProcedure* (ifID, ifVers, objectID, opNum, inParams)
do first Infrag sourceAddress := [sourceAddress] inParams := NULL inParams := [body] seqNum := [seqNum] inFragNum := 0
frag ack *SendPkt* (*fack*, [], seqNum, inFragNum, NULL, sourceAddress)
do callback isFrag := [frag] noFack := [fack] sourceAddress := [sourceAddress] activityID := [activityID] seqNum := [seqNum] *StartCallback* (bootTime, activityID)
send quack inParams := NULL outParams := NULL *SendPkt* (*quack*, [], seqNum, 0, NULL, sourceAddress)

(continued)

Table 7–8. Server FSM Actions (continued)

do next Infrag inParams := inParams ⊕ [body] inFragNum := [fragNum]
quit call *KillApplicationProcedure ()* *send quack*
send nocall *SendPkt* (*nocall*, [], seqNum, 0, NULL, sourceAddress)
send working *SendPkt* (*working*, [], seqNum, 0, NULL, sourceAddress)
free client This is the point at which the implementation is free to delete all information it is keeping about the client activity (activity ID, sequence number, and so on) so that it can optimize the amount of space available to a server that handles multiple clients simultaneously.
send reject outParams := StatustoNDR([rejectStatus] replyType := *reject* *SendPkt* (*reject*, [], seqNum, 0, outParams, sourceAddress)
free reply outParams := NULL
send fault outParams := StatustoNDR([faultStatus] replyType := *fault* *SendPkt* (*fault*, [], seqNum, 0, outParams, sourceAddress)
send first outfrag *setup reply* *SendPkt* (*response*, [frag], seqNum, outFragNum, outParams[outFragNum], sourceAddress)

(continued)

Table 7–8. Server FSM Actions (continued)

send reply *setup reply* replyType := *response* *SendPkt* (*response*, [], seqNum, 0, outParams, sourceAddress)
setup reply outParams := [outParams] inFragNum := –1 replyCount := 0
resend outfrag replyCount++ *SendPkt* (*response*, [frag], seqNum, outFragNum, outParams[outFragNum], sourceAddress)
send next outfrag outFragNum++ *SendPkt* (*response*, [frag], seqNum, outFragNum, outParams[outFragNum], sourceAddress)
send last outfrag outfragNum++ *SendPkt* (*response*, [frag, lastFrag], seqNum, outFragNum, outParams[outFragNum], sourceAddress)
resend reply replyCount++ *SendPkt* (replyType, [], seqNum, outFragNum, outParams[outFragNum], sourceAddress)

7.8 Server Primitive Syntax Descriptions

This section defines the syntax of the primitives used in server FSM actions.

NAME

KillApplicationProcedure — Abort execution of a remote operation.

SYNTAX

KillApplicationProcedure ()

DESCRIPTION

KillApplicationProcedure directs the execution engine to stop executing a remote operation. The server FSM invokes this primitive in response to a client *quit* message.

NAME

SendPkt — Build an NCA/RPC packet and send it over the communications medium.

SYNTAX

SendPkt (pkttype, pktflags, seqnum, fragnum, data, location)

INPUT PARAMETERS

pkttype	A server-initiated packet type (from []). Determines the packet type field in the packet header.
pktflags	One or more flags (from []). Determines the packet flags field in the packet header.
seqnum	An integer. Determines the sequence number field in the packet header. This value is obtained from the [seqNum] portion of the message input to the server FSM.
fragnum	An integer. Determines the fragment number field in the packet header.
data	A variable-length string of bytes. Determines the body portion of the packet. The length of *data* determines the body length field in the packet header.
location	A socket address. Determines the destination address for the packet. This value is obtained from the [sourceAddress] portion of the message input bundle to the server FSM.

DESCRIPTION

SendPkt builds the packet header and body with values specified in its input parameters and sends the packet over the communications medium to the target destination specified in the [sourceAddress] portion of the message input bundle. All packet fields not modifiable by the server must retain the values established by the client FSM.

NAME

StartApplicationProcedure — Execute the operation specified in the client request.

SYNTAX

StartApplicationProcedure (interface, version, object, operation, inparams)

INPUT PARAMETERS

interface A UUID. The universal unique identifier of the interface as specified in the *request* packet header.

version An unsigned long integer. The version number of the interface as specified in the *request* packet header.

object A UUID. The universal unique identifier of the object as specified in the *request* packet header.

operation An unsigned short integer. The number of the operation within the interface as specified in the *request* packet header.

inparams An array of bytes. The data received in the *request* body (the re-assembled data, if the client request was sent in fragments) for input to the operation.

DESCRIPTION

StartApplicationProcedure directs the execution engine to execute the operation requested by the client and to return any output data from that operation.

NAME

StartCallback — Start the callback mechanism.

SYNTAX

StartCallback (request_source, boot_time, activity)

INPUT PARAMETERS

request_source A socket address. The location from which the client request was sent.

boot_time An unsigned long integer. The time the server last booted, as specified in the global variable bootTime.

activity A UUID. The activity identifier contained in the *request* packet header.

DESCRIPTION

StartCallback directs the execution engine to make a remote procedure call to the conv_who_are_you operation in the Conversation Manager running on the host that generated the client request. *StartCallback* supplies as *request_source* the socket address given in the [sourceAddress] portion of the *request* message input bundle; the execution engine uses the value in *request_source* to construct a handle for input to the conv_who_are_you call.